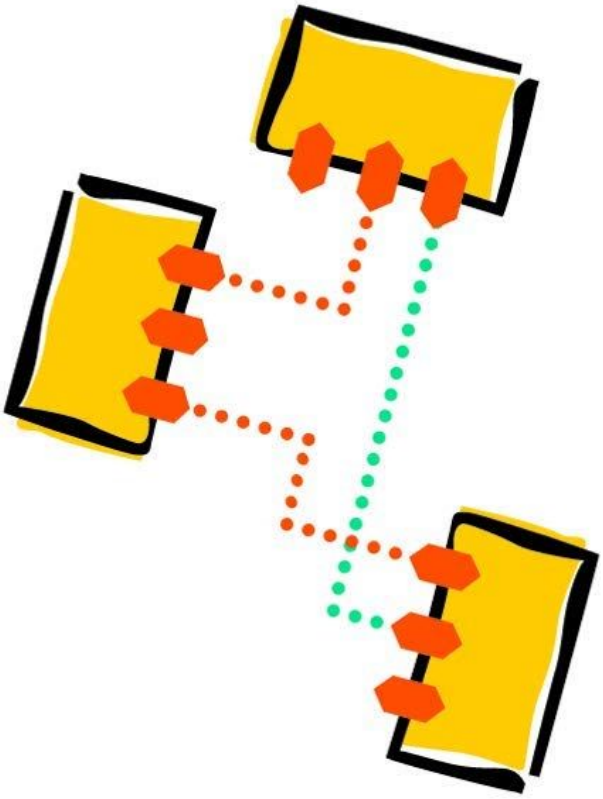
Hanspeter Mössenböck

Objektorientierte Programmierung in Oberon 2

3. Auflage



Springer

Objektorientierte Programmierung in Oberon-2

Springer-Verlag Berlin Heidelberg GmbH

Hanspeter Mössenböck

Objektorientierte Programmierung in Oberon-2

Geleitwort von Niklaus Wirth

Dritte, vollständig überarbeitete und erweiterte Auflage

Mit 156 Abbildungen und CD-ROM

Springer

Prof. Dr. Hanspeter Mössenböck

Johannes Kepler Universität Linz
Institut für Praktische Informatik
Altenbergerstraße 69
A-4040 Linz

Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-3-540-64649-5 ISBN 978-3-642-58985-0 (eBook)
DOI 10.1007/978-3-642-58985-0

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
Objektorientierte Programmierung in Oberon-2
[Medienkombination] / Hanspeter Mössenböck.
Berlin; Heidelberg; New York; Barcelona; Budapest; Hongkong;
London; Mailand; Paris; Singapur; Tokio: Springer, 1998

Buch. – 3. Aufl. – 1998 brosch. CD-ROM zur 3. Aufl. 1998

Dieses Werk (Buch und CD-ROM) ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

© Springer-Verlag Berlin Heidelberg 1998
Ursprünglich erschienen bei Springer-Verlag Berlin Heidelberg New York in 1998

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

SPIN: 10681183 33/3142 543210

Geleitwort von Professor Niklaus Wirth

Ohne Zweifel hat die Idee der Objektorientierung Bewegung in das Feld der Programmiermethodik gebracht und die Menge der Programmiersprachen vergrößert. Zwar ist sie an sich nicht neu, ist sie doch bereits in den 60er Jahren aufgetaucht. Den Anstoß gaben Anwendungen zur Simulation von Systemen mit diskreten Ereignissen. Manifest wurde sie erstmals in der Sprache Simula 67. Fast zwanzig Jahre später kam der Stein aber erst richtig ins Rollen, und heute ist die objektorientierte Programmierung zu einem wichtigen Begriff und zu einer potenten Technik geworden. Man kann zuweilen sogar von einem Überschwingen sprechen, denn der Begriff ist mittlerweile auch als Schlagwort bekannt. Schlagwörter aber tauchen stets dort auf, wo Hoffnung besteht, unkundigen Klienten etwas andrehen zu können, weil sie sich davon die Lösung all ihrer Schwierigkeiten versprechen. Daher wird auch die objektorientierte Programmierung oft als Heil- und Wundermittel angepriesen. Die Frage ist daher berechtigt: Was steckt wirklich dahinter?

Um es vorwegzunehmen: Es steckt mehr dahinter als das vielzitierte in den Vordergrundrücken der Daten als Objekte anstelle der Abläufe (Algorithmen), denen die Daten unterworfen werden. Es handelt sich um mehr als eine rein alternative Betrachtungsweise von programmierten Systemen. Diese Frage ist das zentrale Thema des vorliegenden Buches. Es ist ein Lehrbuch und zeigt in didaktisch geschickter Weise, welche Konzepte und Konstrukte neu sind, wo sie vernünftigerweise eingesetzt werden, und welche Vorteile anfallen. Denn es trifft nicht zu, daß alle Probleme nutzbringend in objektorientierter Weise neu programmiert werden. Im Gegenteil, die neue Methodik kommt eigentlich erst recht zum Tragen, wo komplexe Datenstrukturen und Algorithmen ins Spiel kommen. Es wäre verfehlt, die konventionelle Sicht bereits ad acta zu legen.

Es ist ein Verdienst des Autors, die Konzepte der objektorientierten Programmierung in aufbauender Weise einzuführen, sie in evolutionärer Art darzulegen und an geeigneten Beispielen zu zeigen, wo die

zusätzlichen Konzepte sinnvoll angewendet werden. Dazu bietet die Programmiersprache Oberon-2 eine ausgezeichnete Grundlage, da sie die wenigen, typischen objektorientierten Konstrukte denjenigen der konventionellen, prozeduralen Programmierung hinzufügt. Der Leser möge sich aber stets bewußt sein, daß nicht die Sprache, sondern die Methodik und Disziplin das eigentliche Anliegen ist. Die Sprache dient lediglich der Formulierung in klarer und prägnanter Art. Man spricht daher von der Unterstützung einer Methodik durch eine Sprache; Oberon-2 unterstützt die objektorientierte Methodik.

Wenn das Paradigma der Objektorientierung vor allem bei komplexen Systemen vielversprechend ist, so vor allem deshalb, weil die Technik der objektorientierten Programmierung es gestattet, modulare Systeme echt erweiterbar zu gestalten, und zwar so, daß nicht nur neue Operationen aufbauend auf alten hinzugefügt werden können, sondern daß dasselbe auch für Datentypen und deren Instanzen gilt. Mit diesen Bemerkungen ist bereits angedeutet, daß die Objektorientierung erst in Kombination mit Modularität und strikter Typisierung der Daten einen Durchbruch darstellt.

Das vorliegende Buch stellt eine schön gegliederte Einführung in dieses neue Sachgebiet dar. Es ist offensichtlich, daß sein Autor aus dem Vollen schöpfen kann, weil er sich seit Jahren intensiv mit dem Gebiet befaßt und die Methodik mit Erfolg gelehrt hat. Es ist daher eine Bereicherung für jeden, der an moderner Programmiermethodik interessiert ist.

Niklaus Wirth

Vorwort zur erweiterten dritten Auflage

Seit dem ersten Erscheinen dieses Buches vor 6 Jahren hat sich auf dem Gebiet der objektorientierten Programmierung einiges getan. Daher wurde das Buch anläßlich seiner dritten Auflage gründlich überarbeitet und erweitert.

Zu den fruchtbarsten Ideen der letzten Jahre zählt die Entwicklung objektorientierter *Entwurfsmuster*. Sie haben die Art, wie wir über objektorientierte Programme denken, revolutioniert. Daher enthält die neue Auflage dieses Buches ein ausführliches Kapitel über Entwurfsmuster. Die weiteren Kapitel – insbesondere die Fallstudie Oberon0 – zeigen, wie man diese Muster in die Praxis umsetzt.

In letzter Zeit haben sich auch *objektorientierte Frameworks* stark verbreitet. Man hat erkannt, daß die Vorteile der objektorientierten Programmierung erst dann wirklich zum Tragen kommen, wenn man die Wiederverwendung nicht auf einzelne Klassen beschränkt, sondern auf der Ebene ganzer Frameworks betreibt. Das Kapitel über Frameworks wurde daher in der dritten Auflage ausgebaut.

Objektorientierte Systeme werden heute meist mit *grafischen Notationen* beschrieben. Insbesondere hat sich die OMT-Notation und die daraus entwickelte UML-Notation durchgesetzt. Daher werden die meisten Beispiele dieses Buches nun in UML-Notation beschrieben, was sie auf eine höhere Abstraktionsebene hebt und somit verständlicher macht.

Ein neues Kapitel über *Kontrakte* beschreibt schließlich die formalen Grundlagen der Korrektheit von Klassen und ihrer Erweiterungen. Dieses Wissen ist auch für Praktiker wichtig, da man keine großen objektorientierten Systeme schreiben kann, ohne die Voraussetzungen für ihre Korrektheit zu verstehen.

Der Springer-Verlag hat sich dankenswerterweise bereit erklärt, der neuen Auflage des Buches eine *CD* beizulegen, auf der das komplette Oberon-System sowie die Quellprogramme der Fallstudie aus Kapitel 12 enthalten sind. Obwohl diese Programme auch aus dem Internet

abrufbar sind (siehe Anhang D), werden sich doch viele Leser freuen, wenn sie die Software gleich mit dem Buch mitgeliefert bekommen.

Oberon-2 ist nach wie vor eine sehr moderne Programmiersprache, in der die Konzepte der Objektorientierung in großer Reinheit hervortreten. Insbesondere ist sie hervorragend für die Ausbildung geeignet, da sie einfacher und kleiner als die meisten anderen objektorientierten Sprachen ist. Viele Schulen und Universitäten setzen aus diesem Grund Oberon-2 als Ausbildungssprache ein (siehe Anhang D).

Ich möchte an dieser Stelle meinen Assistenten – insbesondere Markus Hof, Christoph Steindl, Markus Knasmüller und Stefan Chiettini – danken, die das Oberon-System an der Universität Linz warten und weiterentwickeln. Nicht zuletzt danke ich auch den Studenten meiner Vorlesungen, die viele neue und nützliche Werkzeuge zum Oberon-System hinzugefügt haben.

Hanspeter Mössenböck, Linz 1998

Vorwort zur zweiten Auflage

Die zweite Auflage unterscheidet sich inhaltlich kaum von der ersten. Neben einigen Korrekturen wurden vor allem Teile der Sprachdefinition von Oberon-2 präzisiert (Anhang A). Die Sprache selbst hat sich nicht verändert.

Das Oberon-System und Oberon-2 fanden in letzter Zeit reges Interesse – sowohl an Universitäten als auch in der Wirtschaft. In der Schweiz hat sich eine Oberon-Fachgruppe der Schweizer Informatikergesellschaft gebildet, deren Ziel die Verbreitung von Oberon vor allem auch in der Wirtschaft ist. In England befaßt sich eine Standardisierungsgruppe mit Oberon-2. Daneben gibt es nun auch bereits mehrere Firmen, die Oberon-2-Compiler für IBM PC und Workstations anbieten. Auch die ETH hat ihr Angebot an Portierungen erweitert. Es gibt Oberon-2 nun auch für den IBM PC unter Microsoft Windows™ und Windows NT™, für Silicon Graphics Maschinen sowie für Hewlett-Packard PaRISC-Rechner.

Ich möchte an dieser Stelle allen danken, die mich auf Fehler und Verbesserungsmöglichkeiten in der ersten Auflage des Buches hingewiesen haben. Weitere Kommentare sind willkommen.

Hanspeter Mössenböck, Linz 1994

Vorwort

Objektorientierte Programmierung ist heute ein Schlagwort, das einem aus Zeitschriften und Inseraten ins Auge springt. Was verbirgt sich hinter dem Wort "objektorientiert"? Ist es bloß ein Werbeetikett oder steckt dahinter tatsächlich etwas Nützliches, vielleicht gar eine Wunderwaffe?

Um es vorwegzunehmen: Objektorientierte Programmierung ist kein Wundermittel. Entgegen manchen Versprechungen wird mit ihr das Programmieren nicht kinderleicht. Es ist noch immer eine gehörige Portion Können und Erfahrung nötig, vielleicht sogar noch mehr als bei traditionellen Programmiertechniken. Objektorientierte Programmierung hat aber zweifellos ihre Stärken. Sie führt in vielen Fällen zu eleganteren Lösungen, als es mit herkömmlichen Techniken möglich wäre. Sie fördert die Modularität von Software und damit ihre Lesbarkeit und Wartbarkeit, und sie trägt zur Erweiterbarkeit und Wiederverwendbarkeit von Programmen bei.

Das vorliegende Buch richtet sich an Studenten der Informatik sowie an Programmierer aus der Praxis, die sich über neue Software-Entwicklungsmethoden ein Bild machen wollen. Da immer mehr Sprachen mit objektorientierten Eigenschaften "nachgerüstet" werden, richtet sich das Buch auch an alle Programmierer, die die neuen Fähigkeiten ihrer Lieblingssprache besser nutzen möchten.

Ziel des Buches ist es, die Grundlagen der objektorientierten Programmierung, nämlich Klassen, Vererbung und dynamische Bindung zu vermitteln. Dabei kommt es auf die Konzepte an und nicht auf die Eigenheiten einer bestimmten Programmiersprache. Der Leser soll lernen, wofür sich objektorientierte Programmierung eignet, welche Probleme man mit ihr lösen kann und für welche man besser konventionelle Mittel benutzt.

Objektorientiertes Programmieren ist Programmieren im Großen. Man kann zwar die Prinzipien an kleinen Beispielen erklären, aber um die volle Mächtigkeit und Eleganz dieser Technik zu zeigen, muß man große Beispiele bringen. Gerade das fehlt in den meisten Büchern.

Kapitel 11 dieses Buches beschreibt daher den Entwurf und die Implementierung eines genügend großen Systems samt Quellcode.

Für die Beispiele in diesem Buch wurde keine der weit verbreiteten Sprachen wie Smalltalk oder C++ gewählt, sondern Oberon-2, eine Sprache in der Tradition von Pascal und Modula-2. Der Grund für diese Wahl ist, daß Oberon-2 vom Sprachumfang kleiner ist als die meisten anderen objektorientierten Sprachen (sogar kleiner als Pascal) und man daher schnell mit ihr vertraut wird. Objektorientierte Elemente fügen sich harmonisch in die Sprache ein und verdrängen nicht andere bewährte Konstrukte wie Records, Arrays und Prozeduren. Wenn der Leser die in diesem Buch vermittelten Konzepte verstanden hat, sollte es für ihn leicht sein, sie in jede beliebige andere Sprache umzusetzen.

Das Oberon-System wurde in den Jahren 1985 bis 1987 von den Professoren Niklaus Wirth und Jürg Gutknecht an der ETH Zürich entwickelt. Es besteht sowohl aus einem Betriebssystem als auch aus der Sprache Oberon, in deren Design die Erfahrung eines Mannes steckt, der schon Algol W, Pascal und Modula-2 entworfen hat. Ich selbst war an der Entwicklung des Systems nicht beteiligt, war aber einer seiner ersten Benutzer und brachte durch Oberon-2 einige Erweiterungen an der Sprache an, die sie für die objektorientierte Programmierung geeigneter machen.

Das vorliegende Buch ist weder eine allgemeine Einführung in die Programmierung noch ein Handbuch für Oberon-2; diese Aufgabe wird von anderen Büchern wahrgenommen [ReW92, Rei91]. Es wird vorausgesetzt, daß der Leser bereits eine imperative Sprache wie Pascal oder Modula-2 beherrscht. In Kapitel 2 wird Oberon-2 nur so weit beschrieben, wie es für das Verständnis der Beispiele in diesem Buch nötig ist. Anhang A enthält die vollständige Sprachdefinition.

Mein Dank und meine Bewunderung gilt in erster Linie den beiden Autoren von Oberon für den eleganten Entwurf des Betriebssystems und der Sprache sowie für die ergonomische und effiziente Implementierung, die das Arbeiten mit Oberon zu einem Vergnügen macht.

Viele der Beispiele in diesem Buch verdanke ich meinen Assistenten Robert Griesemer, Clemens Szyperski und Josef Templ. Josef Templ hat auch wertvolle Ideen für Oberon-2 beigetragen.

Neben den oben genannten Personen möchte ich nicht zuletzt Prof. Peter Rechenberg, Prof. Jörg R. Mühlbacher, Dr. Martin Reiser, Dr. Günther Blaschek und Dr. Erich Gamma für das sorgfältige Lesen des Manuskripts und für etliche Verbesserungsvorschläge danken.

Hanspeter Mössenböck, Zürich, 1992

Inhaltsverzeichnis

1 Überblick

1.1 Grundideen 1
1.2 Objektorientierte Programme 3
1.3 Objektorientierte Sprachen 6
1.4 Unterschiede zu herkömmlicher Programmierung 9
1.5 Klassen als Abstraktionsmechanismen 11
1.6 Geschichte objektorientierter Sprachen 14
1.7 Zusammenfassung 15

2 Oberon-2

2.1 Merkmale von Oberon-2 18
2.2 Deklarationen 18
2.3 Ausdrücke 21
2.4 Anweisungen 22
2.5 Prozeduren 23
2.6 Module 25
2.7 Kommandos 30

3 Datenabstraktion

3.1 Konkrete Datenstrukturen 35
3.2 Abstrakte Datenstrukturen 38
3.3 Abstrakte Datentypen 41

4 Klassen

4.1 Attribute und Methoden 45
4.2 Klassen und Module 49
4.3 Beispiele 51
4.4 Fragen 54

5 Vererbung

5.1 Typerweiterung 55
5.2 Klassenhierarchien 59
5.3 Zuweisungen zwischen Objekten 60
5.4 Laufzeit-Typprüfungen 64
5.5 Beziehungen zwischen Klassen 67
5.6 Mehrfache Vererbung 68
5.7 Fragen 70

6 Dynamische Bindung

6.1 Meldungsinterpretation 73
6.2 Abstrakte Klassen 75
6.3 Fragen 79

7 Kontrakte

7.1 Klassenassertionen 82
7.2 Subkontrakte 83
7.3 Parameter überschriebener Methoden 87

8 Typische Anwendungen

8.1 Abstrakte Datentypen 89
8.2 Generische Bausteine 91
8.3 Heterogene Datenstrukturen 97
8.4 Austauschbares Verhalten 101
8.5 Anpassung bestehender Bausteine 103
8.6 Halbfabrikate 106
8.7 Zusammenfassung 108

9 Entwurfsmuster

9.1 Motivation 109
9.2 Erzeugende Muster 111
9.2.1 Konstruktor 111
9.2.2 Fabrik 112
9.2.3 Prototyp 114
9.3 Strukturmuster 116
9.3.1 Familie 116
9.3.2 Adapter 117

9.3.3 Kompositum 118
9.3.4 Dekorator 119
9.3.5 Zwilling 123
9.4 Verhaltensmuster 126
9.4.1 Meldungsobjekt 127
9.4.2 Iterator 130
9.4.3 Beobachter 133
9.4.4 Schablonenmethode 135
9.4.5 Kopieren von Objekten 137
9.4.6 Ein/Ausgabe von Objekten 140
9.4.7 Erweiterung eines Systems zur Laufzeit 142

10 Objektorientierter Entwurf

10.1 Aufgabenorientierte Sicht 145
10.2 Objektorientierte Sicht 146
10.3 Wie findet man Klassen? 148
10.3.1 Grundsätzliche Entwurfsüberlegungen 148
10.3.2 Zusätzliche Entwurfsüberlegungen 150
10.3.3 Ableitung von Klassen aus einem Text 151
10.3.4 CRC-Karten 152
10.4 Schnittstellenentwurf 153
10.5 Abstrakte Klassen 155
10.6 Wann sind Klassen sinnvoll? 157
10.7 Häufige Entwurfsfehler 159
10.7.1 Zu viele triviale Klassen 160
10.7.2 Verwechslung von Ist- und Benutzt-Beziehung 161
10.7.3 Verwechslung von Oberklasse und Unterklasse 162
10.7.4 Varianten mit gleicher Struktur 163
10.7.5 Falsches Empfängerobjekt 163
10.7.6 Zu tiefe oder zu flache Klassenhierarchie 164

11 Frameworks

11.1 Frameworks als erweiterbare Systeme 167
11.2 Ein Framework für die Menüauswahl 170
11.3 Das MVC-Schema 173
11.4 Ein Framework für Objekte in Texten 178
11.5 Application Frameworks 180

12 Fallstudie Oberon0

12.1 Das Fenstersystem (Viewers0) 186
12.2 Verteilung von Benutzereingaben (Oberon0) 197
12.3 Ein Texteditor 199
12.3.1 Einfache Texte (AsciiTexts) 200
12.3.2 Texte mit Schriftarten und Elementen (Texts0) 206
12.3.3 Editieren von Text (TextFrames0) 215
12.3.4 Hauptmodul des Texteditors (Edit0) 230
12.4 Ein Grafikeditor 232
12.4.1 Figuren (Shapes0) 232
12.4.2 Editieren von Figuren (GraphicFrames0) 238
12.4.3 Hauptmodul des Grafikeditors (Draw0) 242
12.4.4 Rechtecke als spezielle Figuren (Rectangles0) 243
12.5 Einbettung von Grafiken in Texte (GraphicElems0) 245

13 Kosten und Nutzen

13.1 Nutzen 251
13.2 Kosten 253
13.3 Ausblick 256

A Sprachdefinition Oberon-2

A.1 Einleitung 257
A.2 Syntax 257
A.3 Terminalsymbole 258
A.4 Deklarationen und Sichtbarkeitsbereiche 259
A.5 Konstantendeklarationen 261
A.6 Typdeklarationen 261
A.6.1 Standardtypen 262
A.6.2 Arraytypen 262
A.6.3 Recordtypen 263
A.6.4 Zeigertypen 264
A.6.5 Prozedurtypen 264
A.7 Variablendeklarationen 265
A.8 Ausdrücke 265
A.8.1 Operanden 265
A.8.2 Operatoren 266
A.9 Anweisungen 269
A.9.1 Zuweisungen 269
A.9.2 Prozeduraufrufe 270
A.9.3 Anweisungsfolgen 271
A.9.4 If-Anweisungen 271

A.9.5 Case-Anweisungen 271
A.9.6 While-Anweisungen 272
A.9.7 Repeat-Anweisungen 272
A.9.8 For-Anweisungen 273
A.9.9 Loop-Anweisungen 273
A.9.10 Return- und Exit-Anweisungen 274
A.9.11 With-Anweisungen 274
A.10 Prozedurdeklarationen 275
A.10.1 Formale Parameter 276
A.10.2 Typgebundene Prozeduren 277
A.10.3 Vordeklarierte Prozeduren 278
A.11 Module 280
A.12 Anhänge zur Sprachdefinition 282
A.12.1 Begriffsdefinitionen 282
A.12.2 Syntax von Oberon-2 284
A.12.3 Modul SYSTEM 285
A.12.4 Die Oberon-Umgebung 286

B Bibliotheksmodule

B.1 Modul In 292
B.2 Modul Out 295
B.3 Modul Modules 297
B.4 Modul Types 298
B.5 Modul OS 299

C Glossar 303

D Bezugsquellen 315

Literaturverzeichnis 319

Index 323

1 Überblick

Objektorientierte Programmierung ist eine Modellierungstechnik, mit der große Programme übersichtlicher, sicherer und wartbarer gestaltet werden können. Durch Sprachkonstrukte wie Vererbung und dynamische Bindung wird daneben die Erweiterbarkeit der Programme in einem Maße gefördert, wie das mit traditionellen, prozeduralen Techniken nicht möglich ist.

Dieses Buch gibt eine umfassende Einführung in die Konzepte der objektorientierten Programmierung und zeigt deren Anwendung anhand realistischer Beispiele. Im ersten Kapitel wollen wir der Frage nachgehen, was das Wesen der objektorientierten Programmierung ausmacht, was ihre Vorteile und Anwendungsmöglichkeiten sind und wie sie sich von prozeduraler Programmierung unterscheidet.

1.1 Grundideen

Grundideen

Die objektorientierte Programmierung basiert auf zwei Grundideen:

1. Programme werden in autonome, miteinander kommunizierende Objekte zerlegt, die für bestimmte Aufgabengebiete zuständig sind und diese unabhängig von anderen Objekten bearbeiten.
2. Programme können mit Objekten arbeiten, ohne genau zu wissen, in welcher Variante sie vorliegen.

Beispiel

Betrachten wir als Beispiel für die erste der beiden Ideen einen Texteditor, also ein bereits nicht mehr ganz triviales Softwaresystem. Indem wir den Editor in seine Bestandteile zerlegen und diese als Objekte modellieren (Abb. 1.1), tritt seine Struktur klarer hervor. Die einzelnen Teile werden für sich einfacher verständlich. Der Editor besteht zum Beispiel aus einem Fenster, in dem Text verwaltet wird. Grafische Elemente wie Buttons oder Scrollbars erlauben dem Benutzer, mit dem Editor zu kommunizieren.

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

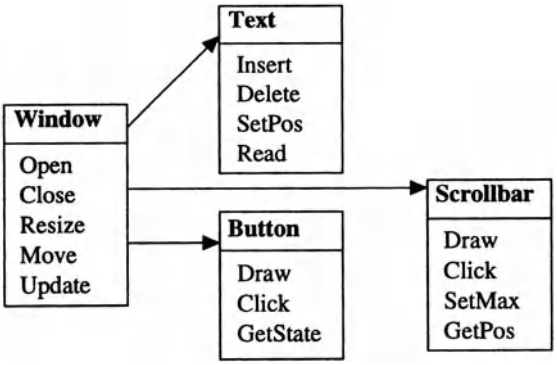


Abb. 1.1 Modellierung eines Editors als Menge selbständiger Objekte

Jedes dieser Objekte hat *private Daten* (ein Fenster hat zum Beispiel eine Größe und eine Position) sowie eine Menge von *Operationen*, mit denen man auf die Objekte zugreifen kann (ein Fenster hat zum Beispiel Operationen zum Öffenen, Schließen, Verschieben, usw.).

Diese Art der Zerlegung nennt man auch *modular* oder *objektbasiert*. Sie ist keine neue Errungenschaft der objektorientierten Programmierung, sondern auch bereits in vielen nicht objektorientierten Sprachen möglich.

Varianten eines Objekts

Richtig objektorientiert wird es erst, wenn man auf ein Objekt Operationen anwenden kann, ohne sich darum kümmern zu müssen, welche *Variante* des Objekts man vor sich hat.

Grafische Benutzeroberflächen kennen zum Beispiel verschiedene Arten von Buttons (Command-Buttons, Radio-Buttons, Checkboxes). Ein Mausklick auf einen Button löst eine bestimmte Reaktion aus, die von der Art des Buttons abhängt. Der Editor möchte aber die verschiedenen Arten von Buttons nicht unterscheiden müssen, sondern einfach auf alle die Operation *HandleClick* anwenden können, wobei jeder Button selbst entscheiden soll, wie er den Klick behandeln will (Abb. 1.2).

Diese Eigenschaft objektorientierter Sprachen werden wir später unter dem Begriff *Polymorphismus* näher kennenlernen. Sie erlaubt es,

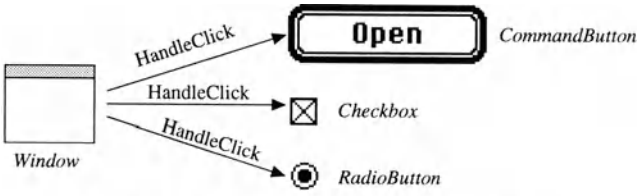


Abb. 1.2 Gleichbehandlung von Varianten

objektorientierte Programme flexibel ändern und erweitern zu können. Man kann zum Beispiel verschiedene Button-Arten gegeneinander austauschen oder später sogar völlig neue Button-Arten hinzufügen, wenn sie nur von außen gleich aussehen wie die alten Buttons.

1.2 Objektorientierte Programme

Schon die äußere Struktur eines objektorientierten Programms unterscheidet sich deutlich von der eines prozeduralen Programms.

Prozedurale Programmstruktur

Ein prozedurales Programm besteht aus einer Hauptprozedur und mehreren davon aufgerufenen Unterprozeduren. Es bildet eine Prozedurenhierarchie wie in Abb. 1.3.

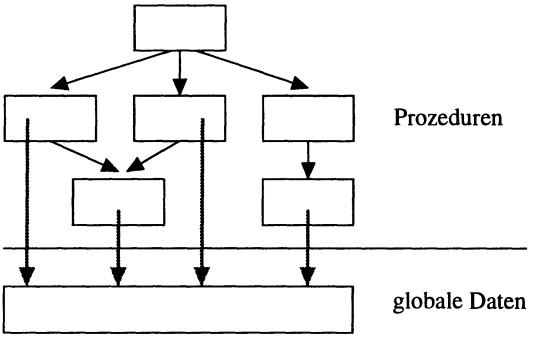


Abb. 1.3 Prozedurale Programmstruktur

Wo bleiben dabei die Daten? Jede Prozedur hat natürlich ihre eigenen lokalen Daten, aber diese leben nur so lange wie die Prozedur aktiv ist. Wirklich wichtige zentrale Daten müssen global gehalten werden und sind daher für alle Prozeduren sichtbar. Das wirft folgende Probleme auf:

- Alle Implementierungsdetails der Daten müssen offengelegt werden, was das Arbeiten mit ihnen oft erschwert.
- Es ist unklar, welche Daten zu welchen Prozeduren gehören. Das macht Programme unübersichtlich und schwer änderbar.
- Die Daten sind nicht vor Zerstörung geschützt. Eine Prozedur kann versehentlich Daten zerstören, auf die sie eigentlich gar keinen Zugriff haben sollte.

Objektorientierte Programmstruktur

Ein objektorientiertes Programm hingegen besteht aus mehreren unabhängigen Objekten (Abb. 1.4). Jedes Objekt ist für einen ganz bestimmten Aufgabenbereich zuständig und enthält nur jene Daten und

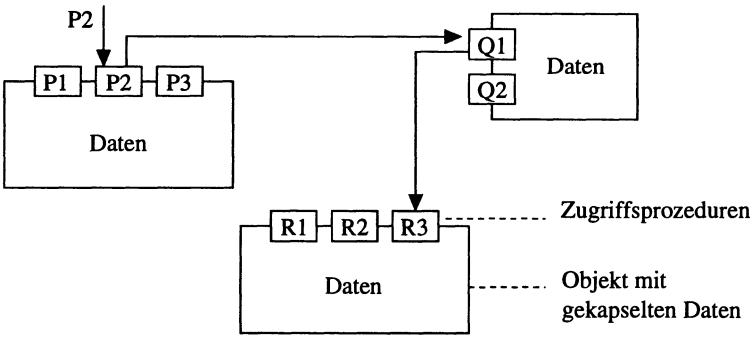


Abb. 1.4 Objektorientierte Programmstruktur

Prozeduren, die nötig sind, um genau diese Aufgaben zu erfüllen. Somit ähnelt es einer dezentral organisierten Firma mit verschiedenen Abteilungen und klar definierten Aufgabenbereichen.

Die Daten eines Objekts sind vor direktem Zugriff von außen geschützt. Andere Objekte können nur über die Prozeduren des Objekts an die Daten heran. Die Prozeduren bilden gewissermaßen eine Schutzschicht um die "gekapselten" Daten. Dies ist wie bei einer gut organisierten Firma. Die Einkaufsabteilung muß nicht wissen, wie die Daten in der Buchhaltung aussehen. Sie überläßt das Buchen getrost der Buchungsabteilung.

Daten und die darauf wirkenden Operationen bilden in objektorientierten Programmen eine Einheit. Man sieht sofort, was zusammengehört. Buchungsoperationen sind zum Beispiel in der Buchhaltungsabteilung zu suchen und sonst nirgends. Dies macht die Programme lesbarer und leichter änderbar.

Die Kapselung der Daten in den Objekten bietet schließlich auch Schutz vor mutwilliger oder ungewollter Zerstörung. Programmierfehler wirken sich so eher lokal aus nicht über weite Bereiche eines Programms hinweg.

Prozedurale Sicht von Operationen

Prozedurale und objektorientierte Programme unterscheiden sich nicht nur in der Programmstruktur, sondern auch in der Art, wie sie Operationen behandeln. In prozeduralen Programmen wird eine Operation durch den Aufruf einer Prozedur ausgedrückt, die Eingabedaten in Ausgabedaten transformiert (Abb. 1.5). Um die Fläche einer Figur *f* zu berechnen, schreiben wir zum Beispiel

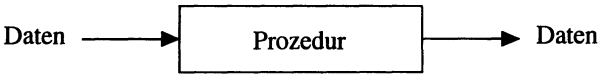


Abb. 1.5 Prozedurale Sicht einer Operation

```
a := Area(f)
```

Die Prozedur *Area* steht dabei im Mittelpunkt, die Daten *a* und *f* eher im Hintergrund.

Diese Sichtweise ist durchaus vernünftig und führt in den meisten Fällen auch zu guten Programmen. Probleme können sich aber ergeben, wenn man in einem Programm mehrere Arten von Figuren hat (z.B. Rechtecke, Dreiecke, Kreise), auf die man alle die Operation *Area* anwenden will. In konventionellen Sprachen kann man dafür nicht dieselbe Prozedur verwenden, sondern braucht für jede Figurenart eine eigene (*RectangleArea*, *TriangleArea*, *CircleArea*).

Noch schlimmer ist, daß man überall, wo man die Fläche einer Figur berechnen will, zwischen den drei Figurenarten unterscheiden muß und dafür zu sorgen hat, daß die richtige Prozedur aufgerufen wird. Im Pseudocode ausgedrückt:

```
IF f is rectangle THEN a := RectangleArea(f)
ELSIF f is triangle THEN a := TriangleArea(f)
ELSIF f is circle THEN a := CircleArea(f)
END
```

Die vielen Fallunterscheidungen blähen den Code auf und bewirken, daß die Figurenarten fest in das Programm eingebrannt sind. Will man später auch Ellipsen behandeln, muß man eine neue Fallunterscheidung einbauen:

```
...
ELSIF f is ellipse THEN a := EllipseArea(f)
...
```

Solche Änderungen müssen an allen Stellen vorgenommen werden, an denen mit Figuren gearbeitet wird. Änderungen dieser Art sind lästig und können leicht vergessen werden.

Objektorientierte Sicht von Operationen

Die objektorientierte Sichtweise stellt hingegen nicht die Prozeduren, sondern die Daten in den Mittelpunkt der Betrachtung. Die Daten und die zu ihnen gehörenden Operationen bilden Objekte, die man auffordern kann, gewisse Aufträge auszuführen und als Ergebnis wieder Daten zu liefern (Abb. 1.6).

Das Besondere daran ist, daß man sich nicht darum kümmern muß,

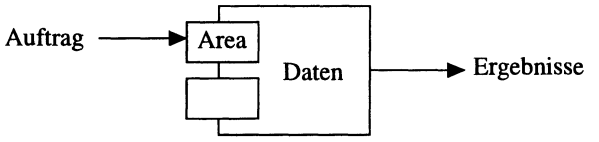


Abb. 1.6 Objektorientierte Sicht einer Operation

von welcher Art das Objekt ist, dem man einen Auftrag erteilt. Jede Objektart interpretiert den Auftrag auf ihre Weise und führt von selbst die richtige Operation aus: Rechtecke interpretieren *Area*, indem sie die Fläche eines Rechtecks berechnen, Kreise, indem sie die Fläche eines Kreises berechnen. Um diese Sicht auszudrücken, benutzt man eine besondere Schreibweise:

```
a := f.Area()
```

bedeutet, daß man der Figur *f* den Auftrag *Area* erteilt. Man sagt auch, man schickt *f* die *Meldung Area*. Dabei ist es gleichgültig, ob *f* ein Rechteck, ein Dreieck oder ein Kreis ist. Selbst wenn später einmal Ellipsen als eine neue Objektart hinzukommen und *f* nun eine Ellipse ist, braucht man die Anweisung *a := f.Area()* nicht zu ändern. Sie funktioniert nach wie vor, solange Ellipsen die Meldung *Area* "verstehen". Die Einführung von Ellipsen läßt also den vorhandenen Code unberührt.

Dieses kleine Beispiel läßt bereits die Vorteile der Objektorientiertheit erahnen: Objektorientierte Programme müssen sich weniger mit Fallunterscheidungen herumschlagen und sind leichter erweiterbar als prozedurorientierte Programme.

1.3 Objektorientierte Sprachen

Die gängigen objektorientierten Sprachen unterscheiden sich in vielen Details, die bei weitem nicht alle für die objektorientierte Programmierung notwendig sind. Welche Eigenschaften muß aber eine Sprache unbedingt aufweisen, um als objektorientiert zu gelten? Im wesentlichen sind das: Unterstützung des Geheimnisprinzips, Datenabstraktion, Vererbung und dynamische Bindung.

Geheimnisprinzip

Das *Geheimnisprinzip* (*information hiding*) besagt, daß man die Implementierung komplexer Daten in einem Baustein verbergen soll. Klienten sollen nur eine abstrakte Sicht davon zur Verfügung gestellt bekommen (Abb. 1.7), d.h. sie können auf die gekapselten Daten nicht direkt zugreifen, sondern nur über Prozeduren, die zum Baustein gehö-

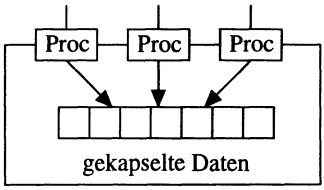


Abb. 1.7 Baustein im Sinne des Geheimnisprinzips

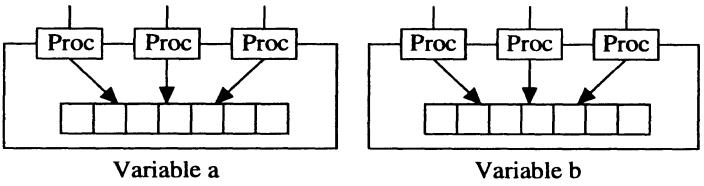


Abb. 1.8 Zwei Variablen *a* und *b* eines abstrakten Datentyps

ren. Dadurch werden sie nicht mit Implementierungsdetails belastet und bleiben von einer eventuellen Änderung der Implementierung unberührt.

Das Geheimnisprinzip wurde bereits 1972 von *David Parnas* propagiert [Par72]. Es wird auch von vielen Sprachen unterstützt, die nicht objektorientiert sind – zum Beispiel in Modula-2 durch Module, in Ada durch Packages.

Abstrakte Datentypen

Abstrakte Datentypen sind eine Weiterführung des Geheimnisprinzips: die oben beschriebenen Bausteine existieren nämlich nur in einem einzigen Exemplar; manchmal möchte man aber mehrere Exemplare davon anlegen (Abb. 1.8).

So, wie man beliebig viele Variablen eines *konkreten* Datentyps *Integer* deklarieren kann, möchte man auch mehrere Variablen eines *abstrakten* Datentyps *Binärbaum* deklarieren können. Und so, wie zu *Integer* die Operationen +, -, * und DIV gehören, sollen zu einem *Binärbaum* Operationen wie Einfügen, Löschen oder Suchen von Elementen gehören.

Integer	+, -, *, DIV, MOD, =, #, <, <=, >, >=
BinaryTree	Insert, Delete, Search, Traverse, ...

Ein abstrakter Datentyp ist also eine Einheit aus Daten und den darauf anwendbaren Operationen. Man kann mehrere Variablen dieses Typs deklarieren. Abstrakte Datentypen sind ebenfalls keine Erfindung objektorientierter Sprachen. Man kennt sie zum Beispiel bereits aus Modula-2 oder Ada.

Vererbung

Vererbung ist hingegen ein Konzept, das man in keiner konventionellen Sprache findet. Es bietet die Möglichkeit, von einem vorhandenen abstrakten Datentyp einen neuen abzuleiten, der alle Eigenschaften des alten Typs erbt, aber zusätzliche Daten und zusätzliche Operationen aufweisen und sogar geerbte Operationen abändern kann. Damit ist es möglich, einen Baustein als Halbfabrikat in einer Bibliothek abzulegen und ihn später zu verschiedenen Endfabrikaten auszubauen. In Abb. 1.9 sind die Typen *Rectangle* und *Circle* aus dem Typ *Figure* abgeleitet. *Circle* erbt zum Beispiel die Attribute *color* und *pattern* und fügt drei neue Attribute *x*, *y*, und *radius* hinzu.

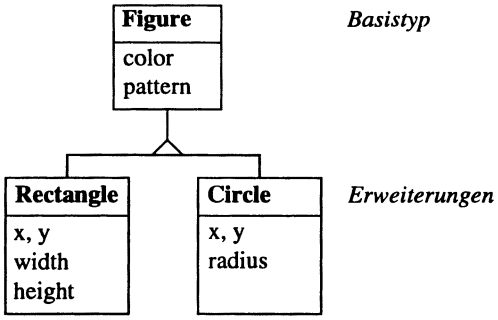


Abb. 1.9 Basistyp *Figure* und verschiedene Erweiterungen

Das Besondere an der Vererbung ist, daß der erweiterte Datentyp mit dem ursprünglichen *kompatibel* bleibt, das heißt, daß alle Programme, die mit Objekten des ursprünglichen Typs arbeiten können, auch in der Lage sind, mit Objekten des neuen Typs zu arbeiten (alle Programme, die mit Figuren arbeiten können, können unverändert auch mit Rechtecken und Kreisen arbeiten). Das erleichtert natürlich die Wiederverwendung vorhandener Algorithmen.

Dynamische Bindung

Das vierte Merkmal objektorientierter Sprachen ist die *dynamische Bindung* von Meldungen (Aufträgen) an Prozeduren: wenn man einem Objekt eine Meldung *Area* schickt, wird erst zur Laufzeit – also dynamisch – entschieden, durch welche Prozedur sie ausgeführt wird.

Wegen der Kompatibilität zwischen einem Basistyp und seinen Erweiterungen kann eine Variable zur Laufzeit nicht nur Objekte desjenigen Typs enthalten, mit dem sie deklariert wurde, sondern auch Objekte beliebiger Erweiterungen davon. Sie kann also *polymorph* (vielgestaltig) sein. In Abhängigkeit vom Objekt, das eine Variable zur Laufzeit enthält, werden Meldungen unterschiedlich ausgeführt.

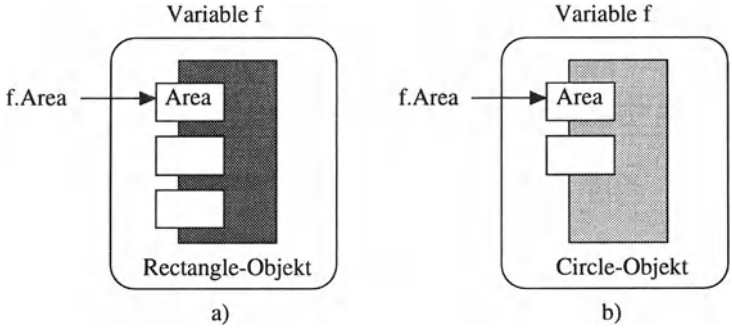


Abb. 1.10 Dynamische Bindung: die Meldung *f.Area* wird durch die *Area*-Prozedur des Objekts ausgeführt, das zur Laufzeit in der Variablen *f* gespeichert ist.

Enthält die Variable *f* ein *Rectangle*-Objekt (Abb. 1.10a), bewirkt *f.Area* den Aufruf der *Area*-Prozedur für Rechtecke; enthält *f* ein *Circle*-Objekt (Abb. 1.10b), bewirkt *f.Area* den Aufruf der *Area*-Prozedur für Kreise.

Dynamische Bindung ist ebenfalls schon seit langem bekannt und zwar in Form von *Prozedurvariablen*: bei Aktivierung einer Prozedurvariablen wird die zur Laufzeit in ihr gespeicherte Prozedur aufgerufen. Das Arbeiten mit Prozedurvariablen ist aber mühsam und fehleranfällig, während die dynamische Bindung in objektorientierten Sprachen eine elegante und sichere Lösung darstellt.

Klassen

Erweiterbare abstrakte Datentypen mit dynamisch gebundenen Meldungen nennt man *Klassen*. Klassen sind die Grundbausteine der objektorientierten Programmierung. Wir werden uns mit ihnen ab Kapitel 4 ausführlich beschäftigen. Zusammenfassend kann man sagen: Objektorientiertes Programmieren heißt Programmieren mit abstrakten Datentypen (Klassen) unter Ausnutzung von Vererbung und dynamischer Bindung.

1.4 Unterschiede zu herkömmlicher Programmierung

Objektorientierte Terminologie

Wenn man zum ersten Mal mit objektorientierter Programmierung in Berührung kommt, fällt einem sofort die ungewohnte Terminologie auf. Anstatt mit Datentypen arbeitet man mit Klassen, anstatt Prozeduren aufzurufen, verschickt man Meldungen. Diese Begriffe wurden von *Smalltalk* [GoR83], einer der ersten objektorientierten Sprachen, geprägt und haben sich eingebürgert, obwohl man – abgesehen von feinen Unterschieden – auch mit der herkömmlichen Terminologie auskommen könnte.

Tabelle 1.1 übersetzt die wichtigsten Begriffe objektorientierter Sprachen in die herkömmliche Terminologie. Die objektorientierten Begriffe sind meist kürzer und griffiger als ihre Übersetzung, weshalb wir sie im folgenden verwenden. Der Leser möge sich aber immer vor Augen halten, daß sie keine grundlegend neuen Konzepte bezeichnen, sondern eine Entsprechung in der traditionellen Begriffswelt haben.

Tabelle 1.1 Objektorientierte Terminologie

Objektorientierte Begriffe	Herkömmliche Begriffe
Klasse	Abstrakter Datentyp (erweiterbar)
Objekt	Wert einer Variablen vom Typ Klasse
Meldung	Prozeduraufruf (dynamisch gebunden)
Methode	Zugriffsprozedur einer Klasse

Ein weiterer Unterschied zwischen objektorientierter und herkömmlicher Programmierung ist die ungewohnte Syntax von Prozeduraufrufen. Um eine Prozedur aufzurufen, die einen Kreis mit der Farbe *color* zeichnet, schreibt man zum Beispiel:

```
circle.Draw(color)
```

Man sagt, daß man dem durch *circle* bezeichneten Objekt (oder einfach dem Objekt *circle*) die Meldung *Draw* schickt. Die Meldung bezeichnet einen *Auftrag* und nicht eine Prozedur. Das Objekt bestimmt selbst, durch welche Prozedur der Auftrag ausgeführt wird. Weil bei dieser Sichtweise das Objekt betont wird, schreibt man *circle* vor den Namen der Meldung.

Diese Unterschiede sind jedoch eher äußerlicher Art. Wichtiger für die objektorientierte Programmierung sind folgende Merkmale:

- Konzentration auf die Daten
- Betonung der Wiederverwendbarkeit
- Programmieren durch Erweitern
- Verteilter Zustand und verteilte Zuständigkeiten

Konzentration auf die Daten

Bei der objektorientierten Programmierung stehen die *Objekte im Mittelpunkt* der Betrachtung und nicht die Prozeduren. Das geht sogar so weit, daß manche Leute fordern, daß es gar keine Prozeduren geben sollte, die nicht irgendeinem Objekt zugeordnet sind. Diese Forderung geht zu weit, denn es gibt durchaus Situationen, in denen das Gewicht stärker auf dem Algorithmus liegt als auf den Daten. Trotzdem sind die Daten oft die Fixpunkte des Entwurfs, an denen die Prozeduren "auskristallisieren".

Betonung der Wiederverwendbarkeit

Objektorientierter Entwurf zielt stärker auf *Wiederverwendung* ab als konventioneller Entwurf. Die meisten Entwurfsmethoden, wie etwa die *schrittweise Verfeinerung* [Wir71], haben zum Ziel, für ein bestimmtes Problem eine genau passende Lösung zu finden. Dabei entstehen maßgeschneiderte Programme, die zwar korrekt und effizient sein können, gegenüber einer Änderung der Anforderungen aber äußerst empfindlich sind. Eine kleine Änderung der Spezifikation kann den gesamten Entwurf über den Haufen werfen.

Beim objektorientierten Entwurf versucht man, die Bausteine nicht auf ihre Klienten zuzuschneiden, sondern sie unabhängig vom gegebenen Kontext zu entwerfen und lieber die Klienten auf die Bausteine abzustimmen. Man trachtet danach, die Bausteine etwas allgemeiner zu machen als es für eine bestimmte Anwendung nötig ist. Das kostet zwar mehr Zeit bei der Entwicklung, macht sich aber später bezahlt:

Man kann die Bausteine dann auch in anderen Programmen verwenden, für die sie ursprünglich gar nicht gedacht waren.

Programmieren durch Erweitern

Objektorientiertes Programmieren heißt oft *Erweitern bestehender Software*. Bausteine wie Fenster, Menüs oder Schalter sind meist als Halbfabrikate in einer Bibliothek vorhanden. Man kann sie erweitern und seinen Bedürfnissen anpassen. Ganze Netze von Klassen können so einer Bibliothek entnommen und zu einem vollständigen Programm ausgebaut werden (siehe Kapitel 11).

Verteilter Zustand und verteilte Zuständigkeiten

In konventionellen Programmen wird der gesamte Programmzustand in den globalen Variablen des Hauptprogramms gespeichert. Das Hauptprogramm ruft zwar Unterprogramme auf, diese haben aber meist keinen Zustand, sondern liefern nur Daten an das Hauptprogramm.

In objektorientierten Programmen ist der Programmzustand auf mehrere Objekte verteilt. Jedes Objekt hat seinen eigenen Zustand (seine eigenen Daten) und besitzt eine Menge von Prozeduren, um ihn zu verwalten. Das Objekt ist nicht bloß für eine einzige Berechnung zuständig, sondern für die Verwaltung eines ganzen Aufgabenbereichs.

Nicht nur der Zustand, sondern auch die Zuständigkeiten sind in objektorientierten Programmen stärker verteilt. Das Hauptprogramm ist weniger wichtig und existiert oft gar nicht. Stattdessen gibt es mehrere (oft gleichberechtigte) Objekt, die miteinander kommunizieren (Abb. 1.11). Ein Objekt weiß zwar, *wofür* andere Objekte zuständig sind, weiß aber nicht, *wie* sie ihre Aufgaben lösen.

1.5 Klassen als Abstraktionsmechanismen

Klassen sind Bausteine, die es erlauben, Dinge der realen Welt in Software zu modellieren. Es ist interessant, ihre Entwicklungsgeschichte in Programmiersprachen zu verfolgen. Die treibende Kraft hinter der Entstehung von Klassen war das Streben nach Abstraktion, d.h. der Wunsch, die semantische Lücke zwischen der problemnahen

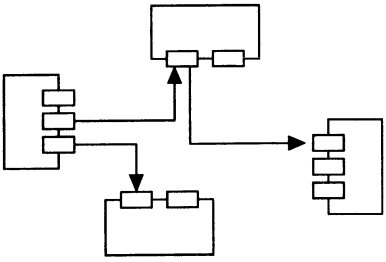


Abb. 1.11 Objekte, die über Meldungen kommunizieren. Jedes Objekt ist für einen bestimmten Aufgabebereich zuständig, den es unabhängig von anderen Objekten verwaltet.

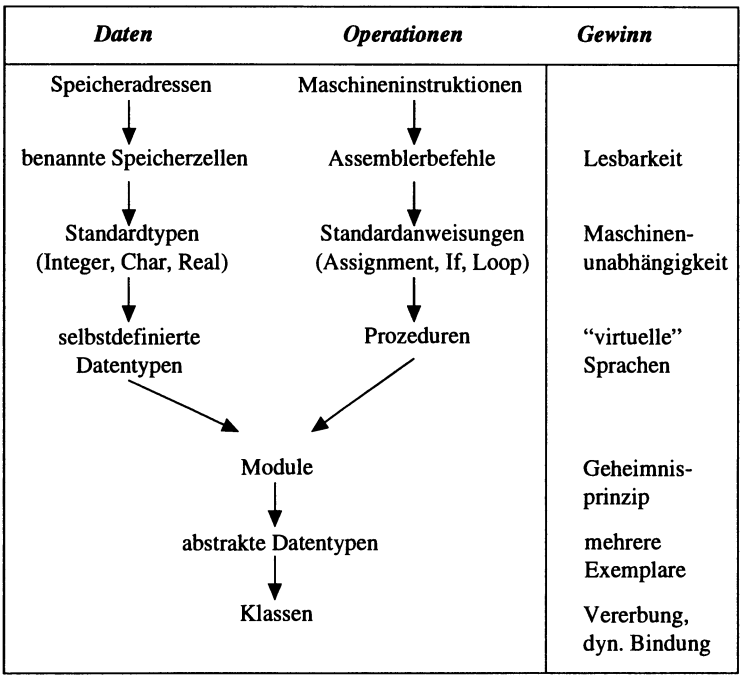


Abb. 1.12 Entwicklung von Abstraktionsmechanismen in Programmiersprachen

Aufgabenstellung und dem maschinennahen Programm zu überbrükken.

Die Entwicklung ging zunächst für Operationen und Daten getrennte Wege. Dann zeigte sich aber doch immer mehr die Tendenz, Daten und dazugehörende Operationen zusammenzufassen. Die objektorientierte Programmierung ist eine logische Folge davon (Abb. 1.12).

Die ersten Rechner kannten als Daten nur unstrukturierte Speicherzellen und als Operationen nur die Befehle der nackten Maschine. Die größte Schwierigkeit bei der Programmierung bestand darin, problemnahe Begriffe wie Kunde oder Konto auf die Maschinenebene abzubilden. Es bestand eine gewaltige Kluft zwischen der Problemwelt und dem Programm.

Variablennamen, Operationsnamen

Die erste Verbesserung wurde mit Assemblern erreicht, die es erlaubten, den Speicherzellen Namen und eine primitive Struktur zu geben. Anstelle binärer Befehlscodes gab es Instruktionsnamen. Das erleichterte das Lesen der Programme, trug aber wenig dazu bei, die semantische Lücke zu beseitigen.

Standardtypen und Standardanweisungen

Erst mit der Einführung höherer Programmiersprachen wie *Fortran* wurde diese Lücke kleiner. Man konnte nun arithmetische Ausdrücke in gewohnter mathematischer Form schreiben, anstatt sie in eine Folge

von Maschinenbefehlen aufzulösen. Die ersten einfachen Datentypen wie *Integer* und *Real* wurden eingeführt, zusammen mit einer Reihe von Operationen, die man auf sie anwenden konnte. Die Datentypen und Operationen waren zwar von der Programmiersprache fest vorgegeben, die durch sie erreichte Maschinenunabhängigkeit bedeutete aber einen gewaltigen Abstraktionsschritt.

Man beachte, daß *Integer* alle Eigenschaften eines abstrakten Datentyps aufweist. Wer *Integer*-Variablen benutzt, braucht nicht zu wissen, ob das höchstwertige Bit links oder rechts steht. Wer die Operation + auf sie anwendet, muß nicht wissen, mit welchem Maschinenbefehl das realisiert wird. Der einzige Unterschied zu abstrakten Datentypen ist, daß der Typ *Integer* bereits von der Programmiersprache vorgegeben ist, während abstrakte Datentypen vom Programmierer definiert werden.

Selbstdefinierte Datentypen und Prozeduren

In den sechziger Jahren wurden Sprachen wie *Pascal* entwickelt. Sie gaben dem Programmierer die Möglichkeit, eigene "virtuelle" Sprachen zu schaffen. Man mußte nicht mehr mit den Datentypen und Operationen der gegebenen Sprache auskommen, sondern konnte sich seine eigenen Datentypen und seine eigenen Operationen in Form von Prozeduren definieren. Diese virtuelle Sprache war der Aufgabenstellung angepaßt und somit problembezogener als eine konkrete Sprache.

Module

Bis hierher entwickelten sich Daten und Operationen getrennt, wenn es auch bemerkenswert ist, daß immer etwa zur gleichen Zeit in beiden Zweigen ähnliche Entwicklungen stattfanden. Zuerst brachten sie bessere Lesbarkeit, dann Maschinenunabhängigkeit und schließlich mehr Problembezogenheit. Ende der siebziger Jahre erkannte man, daß es Vorteile bringt, zusammengehörende Daten und Operationen zu einer Einheit – zu Modulen – zusammenzufassen. Programme wurden dadurch "geordneter". Module – als Sammlungen von Daten und Prozeduren – entsprechen auch besser den Komplexen der realen Welt als einzelne Prozeduren. Module sind heute in modernen Sprachen eine Selbstverständlichkeit. Ohne sie wäre die Entwicklung großer Programme kaum mehr denkbar.

Abstrakte Datentypen

Von Modulen gibt es jedoch immer nur ein einziges Exemplar. Will man mehrere Exemplare eines Bausteins haben, muß man zu abstrakten Datentypen greifen.

Klassen

Durch objektorientierte Sprachen wurde schließlich der Begriff der Klasse eingeführt. Eine Klasse ist ein abstrakter Datentyp, der Vererbung und dynamische Bindung unterstützt. Mit Klassen lassen sich Gegenstände der realen Welt wie Sensoren, Schalter oder Anzeigen direkt in Software modellieren. Die Abstraktionslücke zwischen Problemwelt und Programm ist damit nahezu geschlossen.

1.6 Geschichte objektorientierter Sprachen

Objektorientierte Programmierung ist keineswegs neu. Der Begriff wurde bereits anfangs der siebziger Jahre in Zusammenhang mit *Smalltalk* [GoR83] geprägt, einer Programmiersprache, die in einer Forschungsgruppe der Firma *Xerox* entstand. Die Wurzeln gehen aber noch weiter zurück, nämlich auf die Sprache *Simula* [BDMN79], die 1967 an der Universität Oslo entwickelt wurde und die im wesentlichen bereits alle Merkmale heutiger objektorientierter Sprachen aufwies. Man konnte also bereits Ende der sechziger Jahre objektorientiert programmieren. Es ist darum umso erstaunlicher, daß sich diese Technik erst so spät durchsetzte. Das dürfte wohl damit zusammenhängen, daß Simula und Smalltalk ursprünglich als Spezialsprachen galten: Simula ist eine Simulationssprache und Smalltalk wurde von vielen Fachleuten lange als Spielzeug betrachtet. Erst spät erkannte man den Wert von Klassen für die allgemeine Programmentwicklung.

Smalltalk

Smalltalk wurde zum Inbegriff objektorientierter Programmierung. Auch heute noch ist es eine der konsequentesten objektorientierten Sprachen. Es gibt in ihr keine anderen Datentypen außer Klassen und keine anderen Operationen außer Meldungen. Smalltalk ist allerdings langsam, weil es meist interpretiert wird. Neuere Smalltalk-Systeme erzeugen zwar Maschinencode, die Bearbeitung von Meldungen erfolgt aber auch in diesen Systemen weitgehend interpretativ. Außerdem erlaubt Smalltalk keine statische Typenprüfung und ist daher für die Entwicklung großer Software-Systeme nur bedingt geeignet.

Hybride Sprachen

Mitte der achziger Jahre entstand eine Fülle weiterer objektorientierter Sprachen, meist hybrider Art, das heißt auf bestehende Sprachen wie *Pascal* oder *C* aufgesetzt. In hybriden Sprachen gibt es neben Klassen und Methoden auch gewöhnliche Typen (wie Integer oder Arrays) und Prozeduren. Diese Sprachen erlauben eine Typenprüfung zur Übersetzungszeit. Programme werden in Maschinencode übersetzt und sind daher effizient. Der Umstieg von einer vertrauten Sprache wie C auf einen objektorientierten Dialekt wie C++ ist leicht und hat zur Akzeptanz dieser Sprachen (und damit der objektorientierten Programmierung) beigetragen. Heute gibt es bereits für eine ganze Reihe moderner Sprachen einen objektorientierten Dialekt.

Oberon-2

Die in diesem Buch verwendete Sprache *Oberon-2* ist ebenfalls hybrid. Sie ist sogar noch etwas näher bei traditionellen Sprachen, da es in ihr kein eigenes Klassenkonstrukt gibt. Klassen sind Records, die neben Daten auch Prozeduren enthalten.

1.7 Zusammenfassung

Fassen wir zusammen: Die wichtigsten Merkmale der objektorientierten Programmierung sind

1. Daten und Operationen werden zu Klassen zusammengefaßt, die als Typen für Objekte dienen.
2. Klassen können zu neuen Klassen erweitert werden, die zusätzliche Daten und Operationen enthalten. Objekte einer erweiterten Klasse können überall dort benutzt werden, wo Objekte der Basisklasse zulässig sind.
3. Man bearbeitet Objekte nicht durch Prozeduraufrufe, sondern schickt ihnen Meldungen: man gibt einem Objekt lediglich einen Auftrag und überläßt es dem Objekt, durch welche Prozedur der Auftrag ausgeführt wird. Objekte, die über Meldungen kommunizieren, sind loser gekoppelt als Programmteile, die statisch über Prozeduren zusammenhängen.

2 Oberon-2

Wir verwenden in diesem Buch die Programmiersprache *Oberon-2*, eine objektorientierte Sprache in der Tradition von *Pascal* [JeW74] und *Modula-2* [Wir82].

Das folgende Kapitel soll den Leser mit Oberon-2 bekannt machen. Es ist aber *keine* Einführung in die Programmierung, sondern setzt voraus, daß der Leser bereits programmieren kann. Wer Pascal oder noch besser Modula-2 beherrscht, kann Oberon-2-Programme ohne Schwierigkeiten lesen. Deshalb wird Oberon-2 nur informell anhand einiger Beispiele beschrieben, bevor wir uns dem eigentlichen Thema des Buches – der objektorientierten Programmierung – zuwenden. Der Leser möge Detailfragen aufgrund der Sprachdefinition (siehe Anhang A) klären. Gute Einführungen in das Programmieren mit Oberon-2 findet der Leser in [ReW94], [MLK95] oder [Nik98].

Oberon-2 ist aus *Oberon* entstanden, das – wie schon seine Vorgänger Pascal und Modula-2 – von *Niklaus Wirth* entwickelt wurde [ReW92]. Die Sprache ist einfach und kompakt und eignet sich deshalb hervorragend für die Lehre aber ebenso für die Praxis. Durch die Konzepte der *Typerweiterung* (*Vererbung*) und der *typgebundene Prozeduren* (*Methoden*) ist Oberon-2 eine objektorientierte Sprache.

Oberon-2-Programme laufen üblicherweise unter dem *Oberon-System* – einer *Laufzeitumgebung* mit Kommandoaktivierung, Speicherbereinigung (garbage collection), dynamischem Laden von Modulen und Metainformationen über Programme (siehe [Rei91], [WiG92] sowie Anhang A.12.4). Dies ist ähnlich wie bei Smalltalk- oder Java-Programmen, die ebenfalls eigene Laufzeitumgebungen aufweisen. Es gibt zwar auch eigenständige Oberon-Compiler unter konventionellen Betriebssystemen wie Windows oder Unix, um aber die volle Mächtigkeit von Oberon auszunutzen, sollte man nicht nur die Sprache Oberon-2, sondern auch das Oberon-System verwenden, das meist auf einem Gast-Betriebssystem wie Windows, Unix oder MacOS aufgesetzt ist. Hinweise auf diverse kommerzielle und frei erhältliche Oberon-Systeme findet der Leser in Anhang D.

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

2.1 Merkmale von Oberon-2

Die wichtigsten Merkmale von Oberon-2 sind getrennt übersetzbare Module, strenge statische Typenprüfung, Schnittstellenprüfung von Modulen sowie Objektorientierung.

Module erlauben es, große Programme in kleinere, überschaubarere Teile zu zerlegen, die für sich übersetzt werden können. Dabei stellt der Compiler sicher, daß ihre Schnittstellen zusammenpassen. Man nennt das *getrennte Übersetzung* zum Unterschied von *unabhängiger Übersetzung*, bei der keine Schnittstellenprüfung stattfindet (wie zum Beispiel in C).

Strenge Typenprüfung bedeutet, daß der Compiler bei jeder Operation (z.B. Zuweisung, Addition, Vergleich) prüft, ob die verwendeten Variablen gemäß ihrer Deklaration und somit gemäß der Intention des Programmierers benutzt werden. Auf diese Weise können viele Fehler bereits zur Übersetzungszeit entdeckt werden, was die Kosten ihrer Behebung drastisch senkt.

Die objektorientierten Eigenschaften von Oberon-2 (d.h. Klassen) werden in diesem Kapitel noch ausgeklammert. Sie werden in den Kapiteln 4 bis 6 eingeführt und danach ausgiebig verwendet.

2.2 Deklarationen

Einfache Datentypen

Alle in einem Programm vorkommenden Namen für Konstanten, Typen, Variablen, und Prozeduren müssen vor ihrer Verwendung *deklariert* werden. Dabei wird ihnen ein Datentyp zugeordnet. Die in Tabelle 2.1 angeführten *einfachen Datentypen* sind bereits vordefiniert. Die Wertebereiche dieser Typen sind durch die Sprachdefinition nicht

Tabelle 2.1 Vordefinierte einfache Datentypen in Oberon-2

	Typname	Typischer Wertebereich
Ganze Zahlen	SHORTINT	-128..127
	INTEGER	-32768..32767
	LONGINT	-2147483648..2147483647
Gleitkommazahlen	REAL	± 3.40282E38 (4 Bytes)
	LONGREAL	± 1.79769D308 (8 Bytes)
ASCII-Zeichen	CHAR	0X..0FFX (0..255 hexadezimal)
Boolesche Größen	BOOLEAN	TRUE, FALSE
Mengen	SET	Mengen von Zahlen aus 0..31

festgelegt. Auf den meisten Maschinen gelten aber die in der rechten Spalte von Tabelle 2.1 angegebenen Werte. Eine Deklaration einer ganzzahligen Variablen *i* sieht wie folgt aus:

```
VAR i: INTEGER;
```

Neben den einfachen Datentypen gibt es auch benutzerdefinierte *strukturierte Typen* für Arrays, Records, Zeiger und Prozedurtypen.

Arrays

Ein *Array* ist eine Sammlung von Elementen desselben Typs, die keinen eigenen Namen haben, sondern durch einen Index ausgewählt werden. Beispiele für Arrayvariablen sind:

```
VAR
   a: ARRAY 10 OF CHAR; (* a has 10 elements: a[0], ..., a[9] *)
   b: ARRAY 100, 100 OF INTEGER;
```

Arrays werden mit ganzen Zahlen indiziert. Das erste Element hat den Index 0. Die Elemente werden als *a[i]* und *b[i, j]* angesprochen, wobei zur Laufzeit geprüft wird, ob die Indizes im deklarierten Bereich liegen.

Records

Ein *Record* (Verbund) ist eine Sammlung von benannten *Feldern* beliebigen Typs, zum Beispiel:

```
TYPE
   Person = RECORD
      name: ARRAY 32 OF CHAR;
      idNumber: INTEGER;
      salary: REAL
   END;
```

Wenn *r* eine Variable des Typs *Person* ist, können ihre Felder mit *r.name*, *r.idNumber* und *r.salary* angesprochen werden. Aus Recordtypen kann man neue Typen ableiten (siehe Kapitel 5).

Zeiger

Eine *Zeigervariable* enthält die Adresse eines Records oder Arrays, das dynamisch angelegt wurde, oder den Wert NIL, was bedeutet, daß sie auf kein Record oder Array verweist. Beispiele für Zeigertypen sind:

```
TYPE
   PersonPtr = POINTER TO Person;
   Box = POINTER TO RECORD x, y, width, height: INTEGER END;
   Vector = POINTER TO ARRAY 100 OF INTEGER;
   String = POINTER TO ARRAY OF CHAR;
```

Wenn *p* eine Variable vom Typ *PersonPtr* ist, bedeutet *p^* das (namenlose) Record vom Typ *Person*, auf das *p* zeigt. Auf dessen Feld *name* greift man mit *p^.name* zu. Der Einfachheit halber kann

man das Zeichen ^ auch weglassen und nur *p.name* schreiben. Man abstrahiert dann von der Tatsache, daß *p* nur ein Zeiger auf ein Record ist und nicht das Record selbst. Allerdings muß man sich bewußt sein, daß bei einer Zuweisung *q := p* lediglich *p* zugewiesen wird und nicht *p^*. Der Aufruf der Standardprozedur *NEW(p)* legt Speicherplatz für *p^* an.

Wenn *s* eine Variable vom Typ *String* ist, bedeutet *s^* das Array, auf das *s* zeigt. *s^[i]* bezeichnet das Element mit dem Index *i*. Auch hier kann man das Zeichen ^ weglassen und nur *s[i]* schreiben. *NEW(s, n)* legt Speicherplatz für das Array *s^* mit *n* Elementen an.

Dynamisch erzeugte Speicherbereiche werden in Oberon nie explizit freigegeben. Das Oberon-System verfügt über eine *automatische Speicherbereinigung* (*garbage collector*), die nicht mehr referenzierte Bereiche bei Bedarf einsammelt und wieder verfügbar macht. Das beseitigt eine häufige Fehlerquelle: der Programmierer könnte einen Bereich freigeben, auf den noch ein anderer Zeiger verweist. Ein schreibender Zugriff über diesen Zeiger würde dann einen fremden Speicherbereich zerstören.

Die automatiche Speicherbereinigung ist gerade bei objektorientierten Programmen besonders wichtig. Auf Grund des Geheimnisprinzips kennt der Programmierer die privaten Daten von Objekten nicht. Er kann sich daher nie sicher sein, ob nicht irgendein Objekt noch einen Zeiger auf eine freizugebende Datenstruktur besitzt. Fehler bei der Freigabe von Daten – wie sie zum Beispiel in C++ häufig auftreten – sind äußerst schwer zu finden. Sie lassen sich oft nicht reproduzieren und treten an Stellen auf, die weit von der Stelle ihrer Ursache entfernt liegen. Aus diesem Grund besitzen die meisten modernen objektorientierten Sprachen eine automatische Speicherbereinigung, die diese Fehlerquelle eliminiert.

Prozedurtypen

Variablen eines *Prozedurtyps* (*Prozedurvariablen*) enthalten als Wert eine Prozedur oder NIL (keine Prozedur). Wenn man eine Prozedurvariable aufruft, wird die zu diesem Zeitpunkt in ihr gespeicherte Prozedur aktiviert. Im folgenden Beispiel wird der Prozedurvariablen *write* die Prozedur *WriteTerminal* zugewiesen:

```
VAR write: PROCEDURE (ch: CHAR);

PROCEDURE WriteTerminal (ch: CHAR);
BEGIN
   ...
END WriteTerminal;

write := WriteTerminal;
write(ch);  (*invokes WriteTerminal*)
```

2.3 Ausdrücke

Ausdrücke dienen zur Berechnung von Werten und bestehen aus Operatoren und Operanden. Es gibt vier Arten von Ausdrücken, die in Tabelle 2.2 zusammengestellt sind.

Arithmetische Ausdrücke, Vergleichsausdrücke

Die Bedeutung der arithmetischen Operatoren und der Vergleichsoperatoren ist offensichtlich. In Oberon-2 gelten allerdings weniger strenge Kompatibilitätsregeln als zum Beispiel in Pascal. Insbesondere dürfen numerische Typen (INTEGER, REAL, usw.) in arithmetischen Audrücken gemischt werden. Ferner sind Zeichenarrays miteinander vergleichbar. Die folgenden Beispiele dürften die meisten Fragen beantworten. Die genauen Kompatibilitätsregeln für Ausdrücke sind der Sprachdefinition (Anhang A) zu entnehmen.

```
VAR
   i: INTEGER; j: LONGINT; r: REAL;
   set: SET;
   s: ARRAY 32 OF CHAR;
   sp: POINTER TO ARRAY OF CHAR;
   p, p1: PersonPtr; (*siehe Deklaration im vorigen Abschnitt*)
   proc: PROCEDURE (x: INTEGER);
```

Ausdruck	*Ergebnistyp*
3	SHORTINT
300	INTEGER
100000	LONGINT
0X	CHAR
i + j	LONGINT
i + 3*(r-j)	REAL
i DIV j	LONGINT
i / j	REAL
(s > "John") OR (s = sp^)	BOOLEAN
s = "a"	BOOLEAN
p # p1	BOOLEAN
proc = NIL	BOOLEAN
~ (i IN set)	BOOLEAN

Tabelle 2.2 Arten von Ausdrücken in Oberon-2

	Operatoren	Resultattyp
Arithmetische Ausdrücke	+, -, *, /, DIV, MOD	Numerisch
Boolesche Ausdrücke	&, OR, ~	BOOLEAN
Vergleichsausdrücke	=, #, <, <=, >, >=, IN	BOOLEAN
Mengenausdrücke	+, -, *, /	SET

Boolesche Ausdrücke

Der Ausdruck *~x* bedeutet die Negation von *x*. Die Operatoren & und OR sind *nicht* kommutativ und werden folgendermaßen ausgewertet:

```
a & b     if a then b else false end
a OR b    if a then true else b end
```

Man nennt diese Art der Berechnung *Kurzschlußauswertung* (*short circuit evaluation*), weil die Auswertung des Ausdrucks abgebrochen wird, sobald sein Wert feststeht. Die Kurzschlußauswertung ist für Ausdrücke folgender Art sehr nützlich:

```
IF (p # NIL) & (p.name = "John") THEN ... END
```

Wenn *p* = NIL ist, wird der zweite Teil des Ausdrucks gar nicht mehr berechnet, so daß ein ungültiger Zugriff auf *p.name* vermieden wird.

Mengenausdrücke

Die Mengenoperatoren haben folgende Bedeutung:

```
+  Vereinigung                    {0..7} + {5..9}  = {0..9}
-  Differenz (x-y = x*(-y))       {0..7} - {5..9}  = {0..4}
*  Schnitt                        {0..7} * {5..9}  = {5..7}
/  Symmetrische Differenz         {0..7} / {5..9}  = {0..4, 8..9}
   (x/y = (x-y)+(y-x))
```

Der Ausdruck *i IN s* prüft, ob die Zahl *i* in der Menge *s* enthalten ist.

2.4 Anweisungen

Oberon-2 enthält *elementare Anweisungen* (Zuweisung, Prozeduraufruf, Return, Exit), sowie *zusammengesetzte Anweisungen* für die Auswahl (If, Case) und die Wiederholung (While, Repeat, For, Loop). Die Bedeutung dieser Anweisungen ist so gebräuchlich, daß die folgenden Beispiele genügen sollten. Der Leser möge Details sowie die Bedeutung der Standardprozeduren (ORD, CHR, LEN, etc.) der Sprachdefinition (Anhang A) entnehmen.

```
p.name := "John"                          (*Zuweisung *)
i := 10*i + ORD(ch)-ORD("0")

WriteInt(i, 10)                           (*Prozeduraufruf *)
i := Length(text)

r := p MOD q;                             (*While *)
WHILE r # 0 DO
   p := q; q := r; r := p MOD q
END
```

```
i := 0;                                          (*Repeat*)
REPEAT
    s[i] := CHR(ORD("0") + n MOD 10);
    n := n DIV 10;
    INC(i)
UNTIL n = 0

FOR i := 0 TO LEN(s)-1 DO s[i] := 0X END         (*For*)

i := 0;                                          (*Loop, Exit, If, Return*)
LOOP
    ReadChar(ch);
    IF i = LEN(s) THEN Error; RETURN
    ELSIF ch = 0X THEN EXIT
    END;
    s[i] := ch; INC(i)
END

CASE ch OF                                       (*Case*)
    "a".."z", "A".."Z": ReadIdentifier
|   "0".."9": ReadNumber
|   " ' ", ' " ': ReadString
ELSE ReadSpecial
END
```

Man beachte, daß Zeichenkettenkonstanten einem Zeichenarray fester Länge zugewiesen werden dürfen, falls dieses genügend lang ist, um die Zeichenkette und das Abschlußzeichen 0X aufzunehmen. Das Abschlußzeichen wird automatisch bei der Zuweisung eingefügt. In Oberon muß jede Zeichenkette mit 0X abgeschlossen sein.

Ferner ist zu beachten, daß jede strukturierte Anweisung mit einem Schlüsselwort (meist END) abgeschlossen ist und eine ganze Anweisungs*folge* enthalten kann. Im Gegensatz zu Pascal muß man also die Anweisungsfolge nicht mit BEGIN und END klammern.

2.5 Prozeduren

Auch bei Prozeduren reicht ein Beispiel. Es zeigt eine Prozedur, die eine Zahl *n* in eine Zeichenkette *hex* umwandelt, die die Hexadezimaldarstellung der Zahl enthält.

```
PROCEDURE IntToHex (n: LONGINT; VAR hex: ARRAY OF CHAR);
    VAR i, k: INTEGER; s: ARRAY 8 OF CHAR;

    PROCEDURE Hex (i: LONGINT): CHAR;
    BEGIN (*0 <= i <= 15*)
        IF i < 10 THEN RETURN CHR(i + ORD("0"))
        ELSE RETURN CHR(i-10 + ORD("A"))
```

```
    END
  END Hex;

BEGIN (*IntToHex: assumes n >= 0*)
  i := 0;
  REPEAT s[i] := Hex(n MOD 16); INC(i); n := n DIV 16 UNTIL n = 0;
  k := 0;
  REPEAT DEC(i); hex[k] := s[i]; INC(k) UNTIL i = 0;
  hex[k] := 0X
END IntToHex;
```

Prozeduren bestehen aus einem *Deklarationsteil*, in dem lokale Konstanten, Typen, Variablen und weitere Prozeduren deklariert werden können, und aus einem *Anweisungsteil*, der beim Aufruf der Prozedur ausgeführt wird. Die im Prozedurkopf deklarierten Parameter (*n* und *hex*) nennt man *formale Parameter*. Sie zählen zu den lokalen Variablen der Prozedur. Die beim Aufruf der Prozedur angegebenen Parameter nennt man *aktuelle Parameter*.

Gültigkeitsbereiche

Der *Gültigkeitsbereich* eines Namens, also der Bereich, in dem der Name benutzt werden darf, erstreckt sich von seiner Deklaration bis zum Ende des Blocks (Prozedur oder Modul), in dem er deklariert ist, und überdeckt den Gültigkeitsbereich eines gleichlautenden Namens, der in einem äußeren Block deklariert wurde. Der Gültigkeitsbereich des Parameters *i* in *Hex* überdeckt den Gültigkeitsbereich der Variablen *i* in *IntToHex*. Auf diese Weise kann man in jeder Prozedur beliebige lokale Namen verwenden, ohne sich darum kümmern zu müssen, ob dieselben Namen bereits außerhalb der Prozedur vergeben wurden. Es ist ein Zeichen guten Programmierstils, wenn eine Prozedur nur mit ihren eigenen lokalen Variablen (einschließlich ihrer Parameter) arbeitet und nicht auf globale Variablen oder – noch schlimmer – auf lokale Variablen einer äußeren Prozedur zugreift.

Parameter

In der Prozedur *IntToHex* ist *hex* ein sogenannter *Var-Parameter* (oder *Referenzparameter*), weil er mit dem Symbol VAR deklariert ist. Var-Parameter haben dieselbe Adresse wie der entsprechende aktuelle Parameter, der eine Variable sein muß: wenn man den Wert von *hex* in der Prozedur verändert, verändert man gleichzeitig den aktuellen Parameter. Var-Parameter werden daher für Ausgangsparameter benutzt.

n ist ein *Val-Parameter* (oder *Werteparameter*), weil beim Prozeduraufruf der Wert (value) des aktuellen Parameters nach *n* kopiert wird. Wenn man auf *n* zugreift, arbeitet man mit einer lokalen Kopie; der Wert des aktuellen Parameters bleibt unverändert. Val-Parameter werden für Eingangsparameter benutzt.

Der Typ von *hex* ist ein offenes Array. Die Länge dieses Arrays wird zur Laufzeit bestimmt und ist gleich der Länge des entsprechenden aktuellen Parameters, der ebenfalls ein Array sein muß.

IntToHex ist eine Prozedur, die als Anweisung aufgerufen wird. *Hex* ist hingegen eine *Funktionsprozedur*, die als Teil eines Ausdrucks aufgerufen wird und einen Wert liefert, der in die Berechnung des Ausdrucks einfließt. Den Wert, den eine Funktionsprozedur liefern soll, muß man explizit mit einer *Return-Anweisung* zurückgeben. Funktionsprozeduren erkennt man daran, daß hinter der Liste ihrer formalen Parameter der Typ des Rückgabewerts steht.

Funktionsprozeduren

Prozeduren, die sich selbst aufrufen, nennt man *rekursiv*. Bei jedem Aufruf wird ein neuer Satz lokaler Variablen angelegt, sodaß jede Inkarnation der Prozedur mit ihren eigenen Variablen arbeitet.

Rekursion

Eine Reihe von *Standardprozeduren* wie ORD, CHR, LEN oder COPY sind vordeklariert. Der Leser möge ihre Bedeutung aus Anhang A.10.3 entnehmen.

Standardprozeduren

2.6 Module

Große Programme werden üblicherweise in kleinere Einheiten - sogenannte Module - zerlegt. Ein Compiler besteht zum Beispiel aus einem Parser, einem Scanner, einer Symboltabellenverwaltung und einem Codegenerator (Abb. 2.1). Jedes dieser Module bearbeitet eine abgeschlossene Aufgabe und ist für sich leichter verständlich als der gesamte Compiler.

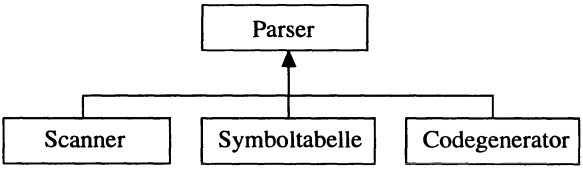


Abb. 2.1 Module eines Compilers.
Pfeile bedeuten die Import-Beziehung.

Ein Modul ist ein Baustein mit einer klar definierten *Schnittstelle*, der benutzt werden kann, ohne zu wissen, wie er implementiert ist, und der implementiert werden kann, ohne zu wissen, in welchem Zusammenhang er später einmal benutzt wird.

Schnittstelle eines Moduls

Im Einklang mit dieser Definition ist ein Modul in Oberon-2 eine Sammlung von Konstanten, Typen, Variablen und Prozeduren, die eine logische und syntaktische Einheit bilden. Seine Schnittstelle besteht aus den Deklarationen derjenigen Namen, die von anderen Modulen benutzt werden dürfen. Man sagt, das Modul *exportiert* diese Namen. Andere Module können das Modul *importieren* und dadurch auf die exportierten Namen zugreifen (siehe später).

Beispiel

Nehmen wir zum Beispiel ein Modul, das die Implementierung eines Wörterbuchs darstellt, in das man Wortpaare eintragen und in dem man Wortpaare suchen kann. Das erste Wort dient dabei als Schlüssel, das zweite als Wert. Die Schnittstelle dieses Moduls könnte folgendermaßen aussehen:

```
DEFINITION Dictionary;
   TYPE String = ARRAY 32 OF CHAR;
   PROCEDURE Clear;
   PROCEDURE Enter (key, value: String);
   PROCEDURE Lookup (key: String; VAR value: String);
   PROCEDURE Print;
END Dictionary.
```

Das Modul exportiert den Typ *String*, sowie die Prozeduren *Clear* zum Löschen des Wörterbuchs, *Enter* zum Eintragen eines neuen Wortpaares, *Lookup* zum Suchen eines Wortes bei gegebenem Schlüssel und *Print* zur Ausgabe des Wörterbuchs auf dem Bildschirm.

Die Schnittstelle ist ein Auszug aus der Implementierung des Moduls. In Oberon-2 muß man sie nicht selbst niederschreiben, sondern zieht sie durch ein Werkzeug (einen sogenannten *Browser*) aus dem Modul heraus (*Browser.ShowDef Dictionary*). Das ist eine große Erleichterung gegenüber Sprachen wie Modula-2, bei denen der Programmierer die Schnittstelle (das sogenannte *Definitionsmodul*) selbst schreiben und mit der Implementierung des Moduls konsistent halten muß.

Die vom Browser erstellte Schnittstellenbeschreibung ist kein Oberon-2-Programm. Sie ist als Dokumentation zu verstehen.

Export

Es folgt nun die Implementierung von *Dictionary*. Alle exportierten Namen sind bei ihrer Deklaration mit einem Stern (*) markiert. Das kennzeichnet sie als exportiert und ermöglicht dem Browser, die Modulschnittstelle zu extrahieren. Der Einfachheit halber wird das Wörterbuch als unsortierte verkettete Liste implementiert.

```
MODULE Dictionary;
IMPORT Out;

TYPE
   String* = ARRAY 32 OF CHAR;
   Node = POINTER TO NodeDesc;
   NodeDesc = RECORD
      key, value: String;
      next: Node
   END;

VAR root: Node;
```

```
PROCEDURE Clear*;
BEGIN
    root := NIL
END Clear;

PROCEDURE Enter* (key, value: String);
    VAR p: Node;
BEGIN
    NEW(p); p.next := root; root := p;
    p.key := key; p.value := value
END Enter;

PROCEDURE Lookup* (key: String; VAR value: String);
    VAR p: Node;
BEGIN
    p := root;
    WHILE (p # NIL) & (p.key # key) DO p := p.next END;
    IF p # NIL THEN value := p.value ELSE value := "" END
END Lookup;

PROCEDURE Print*;
    VAR p: Node;
BEGIN
    p := root;
    WHILE p # NIL DO
        Out.String(p.key); Out.String(" = "); Out.String(p.value); Out.Ln;
        p := p.next
    END
END Print;

BEGIN
    Clear
END Dictionary.
```

Bevor man ein Modul implementiert, überlegt man sich seine Schnittstelle: man schreibt ein *Rumpfmodul*, das nur die Deklarationen der exportierten Namen enthält, wobei die Implementierung von Prozeduren leer bleibt. Für *Dictionary* sieht das folgendermaßen aus:

```
MODULE Dictionary;

    TYPE String* = ARRAY 32 OF CHAR;

    PROCEDURE Clear*;  END Clear;

    PROCEDURE Enter* (key, value: String);  END Enter;

    PROCEDURE Lookup* (key: String; VAR value: String);  END Lookup;

    PROCEDURE Print*;  END Print;

END Dictionary.
```

Damit hat man die Schnittstelle von *Dictionary* festgelegt, die auch der Browser als Definitionsmodul anzeigt. Später füllt man das Rumpfmodul durch weitere Deklarationen und durch Anweisungen aus und gelangt so zur vollständigen Implementierung des Moduls.

Import

Dictionary benutzt zur Ausgabe der Wörter ein Modul *Out*, das am Anfang von *Dictionary* importiert wird. Auch *Out* hat eine Schnittstelle, die auszugsweise folgendermaßen aussieht (die vollständige Schnittstelle von *Out* findet man in Anhang B):

```
DEFINITION Out;
   PROCEDURE String (s: ARRAY OF CHAR);
   PROCEDURE Int (i: LONGINT; w: INTEGER);
   PROCEDURE Ln; (*skip to next line*)
   ...
END Out.
```

In *Dictionary* kann man nun auf alle von *Out* exportierten Namen zugreifen. Dazu muß man sie mit dem Namen des exportierenden Moduls *qualifizieren*. Die Prozedur *Dictionary.Print* enthält zum Beispiel Aufrufe von *Out.String* und *Out.Ln*.

Getrennte Übersetzung mit Schnittstellenprüfung

Ein wesentliches Merkmal von Oberon-2 ist, daß der Compiler die korrekte Verwendung von Modulschnittstellen prüft. Er besorgt sich die Schnittstellenbeschreibung der importierten Module und kennt daher die dort exportierten Namen und ihre Typen. Deshalb kann er sie in Typprüfungen verwenden, wie wenn sie im importierenden Modul selbst deklariert worden wären.

Man nennt das *getrennte Übersetzung* zum Unterschied von *unabhängiger Übersetzung*, bei der Programmteile zwar einzeln übersetzt werden können, der Compiler aber die Schnittstellen nicht prüft (zum Beispiel in Fortran oder C). Die Schnittstellenbeschreibung eines Moduls entsteht bei seiner Übersetzung und wird auf eine sogenannte *Symboldatei* geschrieben.

Bei der Übersetzung von *Dictionary* stellt der Compiler sicher, daß *Out* gemäß seiner Schnittstelle korrekt verwendet wird. Ändert sich die Schnittstelle von *Out*, gilt die Garantie nicht mehr. *Dictionary* muß dann neu übersetzt werden, damit der Compiler die korrekte Verwendung von *Out* erneut prüfen kann. Das Betriebssystem sorgt einstweilen dafür, daß *Dictionary* so lange nicht ausgeführt werden kann, bis es neu übersetzt wurde.

Eine Schnittstellenänderung eines Moduls *M* erzwingt also die Neuübersetzung aller seiner *Klienten* (aller Module, die *M* importieren).

Modulrumpf

Neben Prozeduren kann auch das Modul selbst Code enthalten. Diesen Code nennt man den *Modulrumpf* (die vorletzte Zeile von *Dictionary*). Der Modulrumpf dient vor allem zur Initialisierung der

globalen Daten des Moduls. Er wird ausgeführt, sobald das Modul geladen wird. Vorher werden noch die Rümpfe aller importierten Module ausgeführt, da diese vor dem importierenden Modul initialisiert werden müssen, sonst können sie im Rumpf des importierenden Moduls nicht verwendet werden. Daraus folgt, daß in Oberon-2 keine zyklischen Import-Beziehungen zwischen Modulen erlaubt sind. Die Initialisierungsreihenfolge wäre sonst undefiniert.

Schreibschutz

Beim Export einer Variablen oder eines Recordfeldes kann man angeben, daß Klienten die Variable oder das Feld nur lesen aber nicht verändern dürfen (*read-only*-Zugriff). Das erhöht die Sicherheit des Systems, weil das exportierende Modul sich darauf verlassen kann, daß Klienten seine Daten nicht zerstören. Schreibgeschützte Variablen oder Felder werden mit dem Zeichen "-" anstelle von "*" deklariert. Ein Dateisystem könnte zum Beispiel seine Daten folgendermaßen schützen:

```
MODULE FileSystem;
   TYPE
      File* = POINTER TO FileDesc;
      FileDesc* = RECORD
         name-: ARRAY 32 OF CHAR;
         length-: LONGINT;
         ...
      END;

   VAR resultCode-: INTEGER;
...
END FileSystem.
```

Die beiden Felder *name* und *length* sowie die Variable *resultCode* dürfen von Klienten gelesen aber nicht verändert werden. Nur das exportierende Modul *FileSystem* darf diese Felder und Variablen verändern, weil sie in ihm deklariert sind. Wenn eine strukturierte Variable schreibgeschützt ist, dann gilt das auch für ihre Komponenten: nicht nur *f.name* ist schreibgeschützt, sondern auch *f.name[i]*.

In Sprachen wie C++ oder Smalltalk muß man Informationen, die nicht verändert werden dürfen, durch Zugriffsprozeduren zur Verfügung stellen. Schreibgeschützter Export in Oberon-2 ist eine effizientere Lösung.

Zweck von Modulen

Module sind ein nützliches Sprachmittel, das mehrere Zwecke erfüllt. Sie sind zunächst einmal ein *Strukturierungsmittel*. Sie gruppieren Daten und die dazugehörigen Operationen zu einer Einheit und schaffen damit Ordnung in einem Programm.

Module sind auch ein *Abstraktionsmittel*. Sie verbergen Implementierungsdetails vor anderen Modulen und bieten ihre Dienste über eine einfache Schnittstelle an. Module bilden eine Mauer um ihre privaten

Daten. Die in ihnen deklarierten Namen sind außerhalb nur dann bekannt, wenn man sie exportiert. Die außerhalb deklarierten Namen sind im Modul nur dann bekannt, wenn man sie importiert. Import und Export machen die Kopplung zweier Module sichtbar.

Ein Modul bildet schließlich eine *Übersetzungseinheit*. Sein Quelltext steht in einer eigenen Datei, und der aus ihm erzeugte Objektcode wird ebenfalls in eine eigene Datei geschrieben. Damit sind Module die kleinsten austauschbaren Einheiten in einem System. Der Codegenerator aus Abb. 2.1 kann zum Beispiel ohne Neuübersetzung seiner Klienten gegen einen anderen ausgetauscht werden, nicht jedoch eine einzelne Prozedur des Codegenerators.

2.7 Kommandos

Die bisherigen Ausführungen bezogen sich auf Oberon-2 als *Sprache*; im folgenden Abschnitt geht es nun um Eigenschaften von Oberon als *Arbeitsumgebung*.

Kommandos

In den meisten Betriebssystemen ist die Einheit, die man im Dialog mit dem Computer aufruft, ein Programm. Im Oberon-System ist diese Einheit ein *Kommando*. Als Kommando bezeichnet man jede parameterlose Prozedur *P*, die von einem Modul *M* exportiert wird. In einer typischen Oberon-Umgebung aktiviert man es, indem man seinen Namen (*M.P*) in einem Fenster eintippt und mit der mittleren Maustaste anklickt. Meist steht das Kommando bereits in irgendeinem Fenster und man braucht es nur anzuklicken. Ein Kommando ist also eine *textuelle Schaltfläche* (Button) für die Ausführung einer Aktion. Während grafische Schaltflächen vom Benutzer nur umständlich editiert werden können, kann man Kommandos mit einem gewöhnlichen Texteditor manipulieren. Dadurch kann sich jeder Benutzer seine eigenen Kommandosammlungen anlegen.

Während Prozeduren von anderen Prozeduren aufgerufen werden,

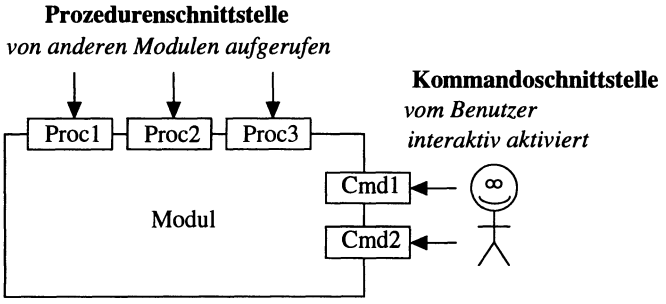


Abb. 2.2 Modul mit Prozeduren- und Kommandoschnittstelle

werden Kommandos vom Benutzer durch Mausklick aktiviert. Ein Oberon-Modul kann somit *zwei* Schnittstellen haben: eine *Prozedurenschnittstelle*, die von anderen Modulen benutzt wird, und eine *Kommandoschnittstelle*, die vom Benutzer interaktiv benutzt wird (Abb. 2.2).

Aktiviert man ein Kommando *M.P*, dann werden das Modul *M* und alle von ihm importierten Module geladen (sofern sie nicht bereits im Speicher stehen). Anschließend wird die Prozedur *P* ausgeführt. Nachdem *P* beendet wurde, bleibt *M* geladen, mit all seinen globalen Daten und ihren Werten. Ruft man anschließend abermals *M.P* (oder ein anderes Kommando aus *M*) auf, wird *M* nicht neu geladen. *P* findet die Werte der globalen Daten noch so vor, wie es sie bei seinem letzten Aufruf hinterlassen hat.

Kommandos können so miteinander über Datenstrukturen im Hauptspeicher kommunizieren und brauchen das nicht über Dateien zu tun. Das ist einfacher und effizienter und ermöglicht zudem, das Wissen über die Struktur der Daten auf das Modul zu beschränken, zu dem die Kommandos gehören.

Beispiel

Wir wollen nun das *Dictionary*-Beispiel aus Kapitel 2.6 so umformen, daß *Clear*, *Enter*, *Lookup* und *Print* vom Benutzer als Kommandos aufgerufen werden können. Die Schnittstelle von *Dictionary* sieht dann so aus:

```
DEFINITION Dictionary;
   PROCEDURE Clear;
   PROCEDURE Enter;
   PROCEDURE Lookup;
   PROCEDURE Print;
END Dictionary.
```

Alle vier Prozeduren sind jetzt Kommandos und können interaktiv aufgerufen werden. Woher bekommen sie aber ihre Parameter?

Kommando-Parameter

Jedes Kommando ist selbst dafür verantwortlich, seine Parameter zu besorgen. Es ist ihm freigestellt, was es als Parameter akzeptiert: den Text, der dem Kommandonamen folgt, den in der jüngsten Selektion enthaltenen Text, den Text an der Einfügeposition oder ein sonstiges markiertes Objekt auf dem Bildschirm. Das Oberon-System stellt geeignete Prozeduren zum Lesen der Parameter zur Verfügung.

In unserem Beispiel entnehmen wir die Parameter dem Text hinter dem Kommando. Der Benutzer aktiviert die Kommandos also als

```
Dictionary.Enter    Buch book
Dictionary.Lookup   Buch
```

Enter nimmt das Wortpaar "Buch book" als Parameter und trägt es ins Wörterbuch ein. *Lookup* nimmt das Wort Buch, schlägt es im Wör-

terbuch nach und liefert das Wort book. Das Kommando *Enter* wird zum Beispiel folgendermaßen implementiert:

```
PROCEDURE Enter*; (*read two words following the command text*)
   VAR p: Node; key, value: ARRAY 32 OF CHAR;
BEGIN
   In.Open; In.Name(key); In.Name(value);
   IF In.Done THEN
      NEW(p); COPY(key, p.key ); COPY(value, p.value);
      p.next := root; root := p
   END
END Enter;
```

Das Modul *In* (siehe Anhang B) stellt Prozeduren zum Lesen von Kommandoparametern zur Verfügung. *In.Open* setzt den Lesezeiger unmittelbar hinter das zuletzt aktivierte Kommando. *In.Name* liest von dort ein Wort bestehend aus Buchstaben und Ziffern. *In.Done* ist TRUE, wenn die vorausgegangenen Leseoperationen erfolgreich waren. Damit hat das Kommando *Enter* seine Parameter und kann wie in Kapitel 2.6 fortfahren.

Man beachte, daß *Dictionary* nach Beendigung von *Enter* geladen bleibt und die Daten des Wörterbuchs somit ihre Werte behalten. Mit neuerlichen Aufrufen von *Enter* können weitere Worte eingetragen, mit *Lookup* können sie gesucht werden.

Entfernen von Modulen aus dem Speicher

Wann wird *Dictionary* aber aus dem Speicher entfernt? In Oberon müssen Module vom Benutzer explizit entfernt werden, falls er das wünscht. Er benutzt dazu ein Kommando, das ihm das Oberon-System zur Verfügung stellt (*System.Free Dictionary* ~). Anschließend kann er eine neue Version von *Dictionary* laden.

Bindender Lader

An dieser Stelle ist zu erwähnen, daß Oberon einen *bindenden Lader* (*linking loader*) besitzt, der Objektmodule erst dann mit anderen Modulen bindet, wenn er sie lädt. Es gibt also keine vorgebundenen Objektdateien, sondern jedes Objektmodul ist eine eigene Datei.

Der Lader sorgt auch dafür, daß jedes Modul nur einmal im Speicher steht. Wenn ein Modul *A* geladen wird, das ein bereits geladenes Modul *B* importiert, dann wird *A* zu diesem geladenen *B* gebunden, und *B* wird nicht neu geladen (Abb. 2.3). Da Module nach ihrem erstmaligen Laden im Arbeitsspeicher bleiben, müssen meist nur wenige Module neu geladen werden. Das verkürzt die Ladezeit und senkt den Speicherbedarf für Oberon-Programme.

Kommandos sind ein nützliches Sprachkonstrukt. Sie erlauben die Herstellung von Programmen mit mehreren Eintrittspunkten. Module können vom Benutzer direkt im Dialog verwendet werden, ohne daß Hauptprogramme geschrieben werden müssen, die die Prozeduren dieser Module aufrufen. Besonders praktisch sind Kommandos bei der Herstellung großer Systeme aus mehreren gleichberechtigten Teilauf-

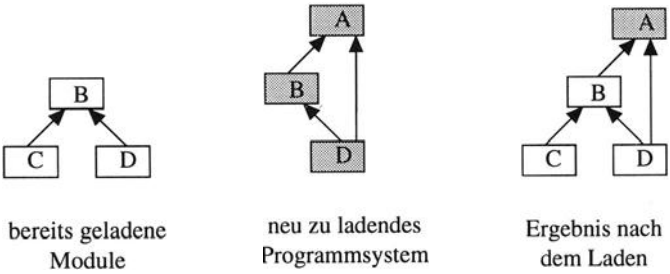


Abb. 2.3 Beim Laden der Module *A*, *B* und *D* wird nur *A* neu geladen. Es wird an die bereits geladenen Versionen von *B* und *D* gebunden.

gaben, wie etwa eines Electronic-Mail- Systems mit den Aufgaben: Mail abschicken, Mail lesen, Mail löschen, etc. Welche dieser Aufgaben sollte zum Hauptprogramm werden und welche sollten ihm untergeordnet sein? Kommandos erlauben es, mehrere Prozeduren gleichberechtigt nebeneinanderzustellen, ohne ihnen ein künstliches Hauptprogramm überzuordnen.

3 Datenabstraktion

Abstraktion ist die wichtigste Waffe im Kampf gegen Komplexität. Sie bedeutet Konzentration auf das Wesentliche und Vernachlässigen von Details. Große Systeme werden nur dadurch verständlich, daß man sie in Bausteine zerlegt, die nach außen hin einfach sind und alle Komplexität in ihrem Inneren verbergen.

Jeder von uns kann ein Fernsehgerät bedienen, ohne die Schaltkreise im Inneren des Geräts zu verstehen. Das wünschen wir uns auch von Software-Systemen: Wir wollen Bausteine mit einer einfachen Schnittstelle, die man benutzen kann, ohne ihr Inneres zu kennen. Mit anderen Worten: Wir wollen von konkreten Datenstrukturen abstrahieren und zu *abstrakten Datenstrukturen* gelangen, oder noch besser zu *abstrakten Datentypen* oder Klassen.

3.1 Konkrete Datenstrukturen

In älteren Programmiersprachen wie Pascal kann man zwar eigene Datentypen definieren, aber ihre Struktur ist dem Programmierer bekannt, ja sie muß sogar bekannt sein, damit er mit diesen Daten arbeiten kann. Man spricht daher von *konkreten Datenstrukturen*.

Prioritätenschlange

Konkrete Datenstrukturen können sehr komplex werden. Betrachten wir als Beispiel eine *Prioritätenschlange* (*priority queue*), in die Elemente eingefügt und nach Prioritäten geordnet wieder entnommen werden können. Der Einfachheit halber nehmen wir an, daß die Elemente Zahlen sind, die gleichzeitig ihre Priorität ausdrücken: je kleiner eine Zahl, desto höher die Priorität. Eine effiziente Datenstruktur zur Implementierung von Prioritätenschlangen ist die *Halde* (*heap*) [Sed88]. Eine Halde ist ein Binärbaum mit n Elementen, die so angeordnet sind, daß der Wert des Vaters immer kleiner oder gleich den Werten seiner Söhne ist. Der Baum ist beinahe balanciert: es gibt eine Zahl h, sodaß alle Blätter die Höhe h oder h-1 haben (Abb. 3.1).

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

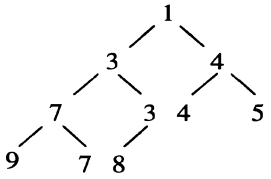


Abb. 3.1 Haldenspeicher mit 10 Elementen

Anders als bei binären Suchbäumen besteht hier keine Ordnung zwischen den beiden Söhnen eines Knotens. Der Wert im linken Sohn kann kleiner, gleich oder größer als der Wert im rechten Sohn sein. Was aber immer gilt, ist, daß der Wert des Vaters kleiner oder gleich den Werten seiner Söhne ist und daß somit die Wurzel des Baums das kleinste Element der Datenstruktur enthält.

Wenn man Abb. 3.1 betrachtet, sieht man, daß alle Ebenen im Baum mit Ausnahme der letzen vollständig gefüllt sind. Die erste Ebene enthält das Element 1, die zweite Ebene die Elemente 3 und 4, die dritte Ebene 7, 3, 4 und 5 und so weiter. Wenn man die Elemente in dieser Reihenfolge in ein Array speichert, ergibt sich das in Abb. 3.2 gezeigte Bild:

0	1	2	3	4	5	6	7	8	9	10
	1	3	4	7	3	4	5	9	7	8

Abb. 3.2 Arraydarstellung des Haldenspeichers aus Abb. 3.1

Diese Implementierung hat den Vorteil, daß man die Zeiger nicht mitzuspeichern braucht, denn die Söhne des Elements *a[i]* befinden sich (falls es sie gibt) in *a[2*i]* und *a[2*i+1]*. Man kann sogar umgekehrt zu jedem Element *a[i]* seinen Vater bestimmen, der sich (falls er existiert) in *a[i DIV 2]* befindet. Die konkrete Datenstruktur eines Haldenspeichers zur Verwaltung von bis zu 127 Zahlen sieht also folgendermaßen aus:

```
VAR
   a: ARRAY 128 OF INTEGER;
   n: INTEGER; (*number of elements in the heap*)
```

Ein neues Element wird eingefügt, indem es an das Ende der Halde (*a[n+1]*) gespeichert und anschließend so lange mit seinem Vater vertauscht wird, bis der Wert des Vaters kleiner oder gleich dem Wert des neuen Elements ist. Wie man aus Abb. 3.3 sieht, ist die Anzahl der Vertauschungen von der Ordnung O(log n).

Folgende Anweisungen fügen ein Element *x* in die Halde *a* ein (in *a[0]* ist der Wert MIN(INTEGER) gespeichert):

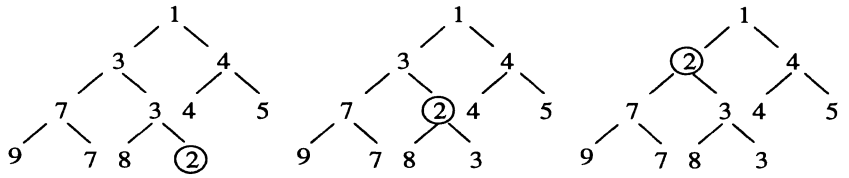


Abb. 3.3 Element 2 wird am Ende der Halde eingefügt und nach oben geschoben, solange es kleiner als das Vaterelement ist

```
n := n + 1; (*virtually insert x at a[n]*)
(*propagate x from a[n] upwards*)
i := n;
WHILE x < a[i DIV 2] DO a[i] := a[i DIV 2]; i := i DIV 2 END;
a[i] := x
```

Einen Haldenspeicher verwendet man für Situationen, in denen man Elemente in sortierter Reihenfolge aus einer Menge entnehmen möchte: das kleinste Element zuerst, das nächstkleinere danach und so weiter. Eine typische Anwendung ist eine Menge von Prozessen, die man nach Zeiten oder Prioritäten geordnet verarbeiten möchte.

Das kleinste Element findet man immer in *a[1]*. Wenn man es entfernt, muß man die Halde jedoch umordnen. Dies geschieht, indem man das letzte Element *a[n]* nach *a[1]* speichert und so lange mit dem kleineren seiner Söhne vertauscht, bis es kleiner als beide Söhne ist oder zu einem Blatt wird (Abb. 3.4). Folgendes Codestück entfernt das kleinste Element *x* aus der Halde *a*:

```
x := a[1];
(*propagate a[n] from a[1] downwards*)
y := a[n]; n := n - 1; i := 1; ready := FALSE;
WHILE (i <= n DIV 2) & ~ ready DO
    j := i + i;
    IF (j < n) & (a[j] > a[j+1]) THEN j := j + 1 END;
    IF y <= a[j] THEN ready := TRUE ELSE a[i] := a[j]; i := j END
END;
a[i] := y
```

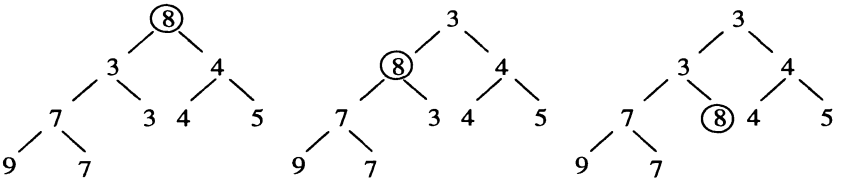


Abb. 3.4 Das Element *a[1]* wurde entnommen. Das Element *a[n]* = 8 wurde nach *a[1]* verschoben und im Baum entsprechend nach unten bewegt

Die Halde *a* und die Anzahl ihrer Elemente *n* bilden in diesem Beispiel die konkrete Datenstruktur der Prioritätenschlange. Die direkte Benutzung der konkreten Datenstruktur in Klienten ist nicht zu empfehlen. Sie hat folgende Nachteile:

Klienten werden mit Details belästigt

Die Klienten müssen die Deklaration der Datenstruktur sowie die Algorithmen zum Einfügen und Entfernen von Elementen kennen. Das macht das Arbeiten mit den Daten kompliziert und belastet die Klienten mit unnötigen Details. Dazu kommt, daß oft der Code für die Zugriffsalgorithmen in *mehreren* Klienten vorhanden ist und daß durch Programmierfehler die Konsistenz der Daten (die Haldenordnung) zerstört werden kann.

Änderungen wirken sich auf Klienten aus

Das Arbeiten mit konkreten Datenstrukturen hat ferner den Nachteil, daß sich Änderungen an den Daten auf deren Klienten (d.h. die benutzenden Programme) auswirken. Wenn man zum Beispiel die Prioritätenschlange nicht mehr als Array fester Länge, sondern als Baum implementieren möchte, in dem beliebig viele Elemente gespeichert werden können, dann ändern sich die Zugriffsalgorithmen, und alle Klienten müssen angepaßt werden. Das ist unangenehm, denn oft weiß man gar nicht, wo die Datenstruktur überall benutzt wird, wer also die Klienten sind. Man kann leicht einen vergessen.

Eigentlich interessiert es die Klienten gar nicht, wie die Prioritätenschlange implementiert ist. Sie wollen sie einfach als Baustein benutzen, dessen Inneres sie nicht zu kennen brauchen. Sie wollen vor allem nicht durch eine Änderung der Baustein-Implementierung belästigt werden. Die konkrete Datenstruktur muß also versteckt werden.

3.2 Abstrakte Datenstrukturen

Abstrakte Datenstrukturen

Eine abstrakte Datenstruktur (ADS) ist ein Baustein bestehend aus Daten und Prozeduren. Die Daten sind im Inneren des Bausteins verborgen und können nur über Zugriffsprozeduren bearbeitet werden (Abb. 3.5). Die Datenstruktur heißt *abstrakt*, weil man nur ihren *Namen* und ihre *Schnittstelle* kennt, aber nicht ihre Implementierung. Man sieht nicht ins Innere des Bausteins hinein.

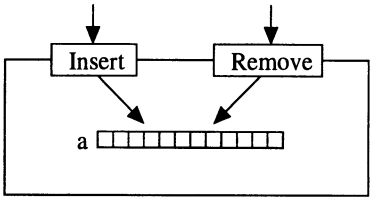


Abb. 3.5 Abstrakte Datenstruktur

Abstrakte Datenstrukturen gehorchen dem *Geheimnisprinzip* (*information hiding*) [Par72]. Alle änderungsanfälligen Daten und Algorithmen sind vor ihren Klienten verborgen. Ihre Implementierung wird als Geheimnis betrachtet und hinter einer Schnittstelle versteckt, die unverändert bleibt, selbst wenn sich die dahinter stehende Implementierung ändert.

Geheimnisprinzip

Abstrakte Datenstrukturen besitzen einen *Zustand*, der durch Zugriffsprozeduren verändert werden kann. Der Zustand drückt sich in den Werten der internen Daten aus. Er bildet sozusagen das „Gedächtnis" für Werte, die von Prozeduraufruf zu Prozeduraufruf erhalten bleiben sollen.

Zustand

In Oberon-2 werden abstrakte Datenstrukturen als Module implementiert. Daten können vor Klienten verborgen werden, indem man sie nicht exportiert. Die Prioritätenschlange wird also zu einem Modul *PriorityQueue* mit folgender Schnittstelle:

Implementierung durch Module

```
DEFINITION PriorityQueue;
   VAR n-: INTEGER; (*number of elements*)
   PROCEDURE Insert (x: INTEGER);
   PROCEDURE Remove (VAR x: INTEGER);
   PROCEDURE Clear;
END PriorityQueue.
```

Insert fügt ein Element ein, *Remove* entfernt das kleinste Element und *Clear* leert die Schlange. Die Anzahl der Elemente wird nicht durch eine Zugriffsprozedur, sondern direkt als Variable *n* geliefert. Es ist unwahrscheinlich, daß sich ihre Implementierung ändert, daher muß man ihren Wert nicht hinter einer Zugriffsprozedur verstecken. Klienten dürfen diese Variable allerdings nur lesend benutzen, weil sie sonst die Korrektheit des Moduls zerstören könnten. Deshalb wird sie mit Schreibschutz exportiert. Die Implementierung von *PriorityQueue* sieht folgendermaßen aus:

Prioritätenschlange als abstrakte Datenstruktur

```
MODULE PriorityQueue;
CONST
   length = 128;
VAR
   n-: LONGINT; (*number of elements*)
   a: ARRAY length OF INTEGER;

PROCEDURE Clear*;
BEGIN
   n := 0, a[0] := MIN(INTEGER)
END Clear;

PROCEDURE Insert* (x: INTEGER);
   VAR i: INTEGER;
BEGIN
```

```
    IF n < length - 1 THEN
        n := n + 1; i := n;
        WHILE x < a[i DIV 2] DO a[i] := a[i DIV 2]; i := i DIV 2 END;
        a[i] := x
    END
END Insert;

PROCEDURE Remove* (VAR x: INTEGER);
    VAR y, i, j: INTEGER; ready: BOOLEAN;
BEGIN
    IF n > 0 THEN
        x := a[1]; y := a[n]; n := n - 1; i := 1; ready := FALSE;
        WHILE (i <= n DIV 2) & ~ ready DO
            j := i + i;
            IF (j < n) & (a[j] > a[j+1]) THEN j := j + 1 END;
            IF y <= a[j] THEN ready := TRUE ELSE a[i] := a[j]; i := j END
        END;
        a[i] := y
    END
END Remove;

BEGIN
    Clear
END PriorityQueue.
```

Die Implementierung der Daten und der Zugriffsalgorithmen ist nun verborgen. Klienten sehen in *PriorityQueue* einen geschlossenen Baustein, einen "schwarzen Kasten" (black box), der einfach zu bedienen ist. Diese Lösung hat mehrere Vorteile:

Vorteile

1. Klienten brauchen die Implementierung von *PriorityQueue* nicht zu kennen, was die Benutzung erleichtert.
2. Die Implementierung kann jederzeit geändert werden, ohne Klienten anpassen zu müssen. Wenn *a* nicht mehr als Array, sondern als Baum implementiert wird (Abb. 3.6), merken die Klienten gar nichts davon, solange die Schnittstelle von *PriorityQueue* gleich bleibt.
3. Die Daten sind im Modul *PriorityQueue* gekapselt und gegen ungewollte Zerstörung gesichert.

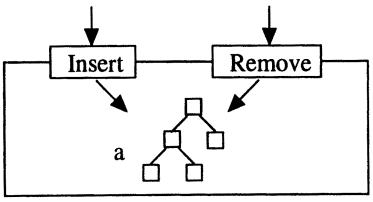


Abb. 3.6 Prioritätenschlange mit geänderter Implementierung aber gleicher Schnittstelle

Datenabstraktion hat aber auch Nachteile:

Nachteile

1. Die Benutzung von *PriorityQueue* ist nicht mehr so effizient wie die einer konkreten Datenstruktur, weil die Zugriffe zu den Daten jetzt über Prozeduren erfolgen. Allerdings sind die Kosten eines Prozeduraufrufs gering.
2. Man kann mit den Daten nur noch jene Operationen ausführen, die in der Schnittstelle vorgesehen sind. Will man später einmal in der Prioritätenschlange nach einem bestimmten Element suchen, dann geht das nicht, weil eine entsprechende Zugriffsprozedur fehlt.

Das Geheimnisprinzip sollte immer mit Bedacht angewendet werden und nie um des reinen Prinzips willen. Wenn man aus Prinzip alle Daten versteckt, kann dadurch sogar die Einfachheit, die Flexibilität und die Erweiterbarkeit eines Bausteins leiden. Man muß sich immer vor Augen halten, was das eigentliche Ziel ist: die Benutzung eines Bausteins so einfach wie möglich zu machen und spätere Änderungen vor Klienten zu verbergen. Das Modul *PriorityQueue* wäre kaum einfacher geworden, wenn man *n* nicht als Variable, sondern als Zugriffssprozedur exportiert hätte. Es geht nicht darum, daß die Klienten private Daten nicht kennen *dürfen*, sondern daß sie sie nicht kennen *müssen*, um mit einem Baustein arbeiten zu können.

3.3 Abstrakte Datentypen

Abstrakte Datentypen

Von abstrakten Datenstrukturen gibt es nur ein einziges Exemplar. Wenn man aber mehrere Exemplare braucht, muß man *abstrakte Datentypen* benutzen. Ein abstrakter Datentyp (ADT) ist ebenfalls ein Baustein aus Daten und Prozeduren, der aber im Gegensatz zu einer abstrakten Datenstruktur als Typ verwendet werden kann. Man kann mehrere Variablen dieses Typs deklarieren.

In Oberon-2 implementiert man abstrakte Datentypen als Records, deren Felder einzeln versteckt werden können, indem man sie nicht exportiert. Die Prioritätenschlange aus unserem Beispiel sieht als abstrakter Datentyp folgendermaßen aus:

```
DEFINITION PriorityQueues;
   TYPE
      Queue = RECORD
         n-: INTEGER (*number of elements*)
      END;

   PROCEDURE Insert (VAR q: Queue; x: INTEGER);
```

```
    PROCEDURE Remove (VAR q: Queue; VAR x: INTEGER);
    PROCEDURE Clear (VAR q: Queue);
END PriorityQueues.
```

Queue ist ein Record, das als Felder die Daten der Prioritätenschlange enthält. Von diesen Feldern wird *n* (schreibgeschützt) exportiert, die anderen Felder sind verborgen (nicht exportiert). Man beachte, daß jede Variable vom Typ *Queue* einen eigenen Satz von Daten hat.

Die Zugriffsprozeduren besitzen einen zusätzlichen Parameter *q* vom Typ *Queue*. Er bezeichnet das Record, auf das sich die Prozeduren beziehen. Weil die Daten der Prioritätenschlange dabei verändert werden, muß *q* ein Var-Parameter sein. Die Implementierung von *PriorityQueues* sieht folgendermaßen aus:

Prioritätenschlange als abstrakter Datentyp

```
MODULE PriorityQueues;
CONST length = 128;
TYPE
    Queue* = RECORD
        n-: LONGINT; (*number of elements*)
        a: ARRAY length OF INTEGER
    END;

PROCEDURE Clear* (VAR q: Queue);
BEGIN q.n := 0, q.a[0] := MIN(INTEGER)
END Clear;

PROCEDURE Insert* (VAR q: Queue; x: INTEGER);
    VAR i: INTEGER;
BEGIN
    IF q.n < length - 1 THEN
        q.n := q.n + 1; i := q.n;
        WHILE x < q.a[i DIV 2] DO q.a[i] := q.a[i DIV 2]; i := i DIV 2 END;
        q.a[i] := x
    END
END Insert;

PROCEDURE Remove* (VAR q: Queue; VAR x: INTEGER);
    VAR y, i, j: INTEGER; ready: BOOLEAN;
BEGIN
    IF q.n > 0 THEN
        x := q.a[1]; y := q.a[n]; q.n := q.n - 1; i := 1; ready := FALSE;
        WHILE (i <= q.n DIV 2) & ~ ready DO
            j := i + i;
            IF (j < q.n) & (q.a[j] > q.a[j+1] THEN j := j + 1 END;
            IF y <= q.a[j] THEN ready := TRUE ELSE q.a[i] := q.a[j]; i := j END
        END;
        q.a[i] := y
    END
END Remove;

END PriorityQueues.
```

Klienten können nun mehrere *Queue*-Variablen anlegen, z.B:

```
VAR negNumbers, posNumbers: PriorityQueues.Queue;
```

und können sie getrennt benutzen:

```
PriorityQueues.Clear(negNumbers);
PriorityQueues.Clear(posNumbers);
...
IF x < 0 THEN PriorityQueues.Insert(negNumbers, x)
ELSE PriorityQueues.Insert(posNumbers, x)
END
```

Erweiterung der Sprache um einen neuen Datentyp

Der abstrakte Datentyp *Queue* kann wie ein konkreter Datentyp (z.B. INTEGER) verwendet werden, um Variablen dieses Typs zu deklarieren. Man hat die Sprache um einen neuen Datentyp erweitert und sie damit für die Lösung eines bestimmten Problems geeigneter gemacht.

Allerdings sind abstrakte Datentypen etwas ineffizenter als abstrakte Datenstrukturen, weil bei jeder Operation das Objekt übergeben werden muß, auf das sich die Operation bezieht. Man sollte sich daher überlegen, wann man abstrakte Daten*typen* (also mehrere Variablen eines Typs) braucht und wann man mit einer abstrakten Daten*struktur* auskommt. Beispiele für abstrakte Datentypen sind *Stack*, *Queue*, *Set*, *File*, *Window* oder *Text*. Hingegen genügen für *Mouse* oder *Terminal* meist abstrakte Datenstrukturen, weil es von ihnen in der Regel nur ein einziges Exemplar gibt.

Abstrakte Datentypen als Zeiger

Abstrakte Datentypen werden oft nicht als Records, sondern als Zeiger auf Records implementiert. Auch in diesem Fall können einzelne Recordfelder versteckt werden. Die Schnittstelle der Prioritätenschlange sieht dann so aus:

```
DEFINITION PriorityQueues1;
   TYPE
      Queue = POINTER TO QueueDesc;
      QueueDesc = RECORD
         n-: INTEGER; (*number of elements*)
      END;
   PROCEDURE Insert (q: Queue; x: INTEGER);
   PROCEDURE Remove (q: Queue; VAR x: INTEGER);
   PROCEDURE Clear (q: Queue);
END PriorityQueues1.
```

Der Parameter *q* kann hier ein Val-Parameter sein, weil nicht er selbst von den Prozeduren verändert wird, sondern nur die Felder des Records, auf das er zeigt.

Klassen als erweiterbare abstrakte Datentypen

Objektorientierte Sprachen bieten für abstrakte Datentypen ein besonderes Sprachkonstrukt an, nämlich *Klassen*. Klassen sind erweiterbare abstrakte Datentypen mit einer speziellen Notation. Sie bilden die Grundbausteine der objektorientierten Programmierung. Wir werden uns in den nächsten Kapiteln intensiv mit ihnen befassen.

4 Klassen

Klassen sind die Grundbausteine der objektorientierten Programmierung. Wie abstrakte Datentypen bestehen sie aus Daten und Operationen. Im Gegensatz zu abstrakten Datentypen sind sie jedoch erweiterbar (siehe Kapitel 5) und werden durch ein spezielles syntaktisches Konstrukt beschrieben, das klar ausdrückt, welche Daten und welche Operationen zusammengehören.

4.1 Attribute und Methoden

Eine Klasse besteht aus ihrem *Namen*, einer Menge von *Attributen* (ihren Datenfeldern) und einer Menge von *Methoden* (ihren Operationen). Zu ihrer Darstellung benutzt man oft eine grafische Notation, die von den syntaktischen Details abstrahiert und sprachunabhängig ist. Als Standard hat sich die *UML-Notation* eingebürgert ([RBJ98], [Fow97]), die wir in der Folge auch verwenden.

Klasse Counter

Abb. 4.1 zeigt zum Beispiel eine Klasse *Counter*, die einen Zählerbaustein repräsentiert. Sie besitzt ein Attribut *value*, das den Wert des Zählers darstellt und zwei Methoden *Add* und *Clear*. *Add(x)* erhöht den Zählerwert um *x*, *Clear* setzt ihn auf den Wert 0 zurück. Je nach gewünschtem Detaillierungsgrad kann man in der Grafik die Typen der Attribute oder Methodenparameter angeben oder auch nicht. In einer völlig abstrakten Form kann man die Methodenparameter sogar ganz weglassen.

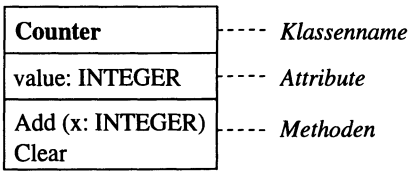


Abb. 4.1 Grafische Darstellung einer Klasse *Counter*

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

Klassen sind Datentypen und können zur Deklaration von Variablen verwendet werden. Jede Variable vom Typ *Counter* enthält ein *Counter*-Objekt mit einem eigenen Attribut *value*, das mit den Methoden *Add* und *Clear* manipuliert werden kann.

Deklaration von Klassen

Soweit entsprechen Klassen gewöhnlichen abstrakten Datentypen. In Oberon-2 (wie auch in anderen objektorientierten Sprachen) benutzt man jedoch eine spezielle Schreibweise, um die Zugehörigkeit der Methoden zur Klasse deutlicher zu machen. Die Klasse *Counter* sieht zum Beispiel in Oberon-2 folgendermaßen aus:

```
TYPE
   Counter = RECORD
      value: INTEGER
   END;

PROCEDURE (VAR c: Counter) Add (x: INTEGER);
BEGIN
   c.value := c.value + x
END Add;

PROCEDURE (VAR c: Counter) Clear;
BEGIN
   c.value := 0
END Clear;
```

Typgebundene Prozeduren

Die Klasse selbst wird als Recordtyp dargestellt, ihre Attribute als Recordfelder. Für die Implementierung der Methoden verwendet man sogenannte *typgebundene Prozeduren*, die sich von gewöhnlichen Prozeduren dadurch unterscheiden, daß sie einen speziellen Parameter vor dem Prozedurnamen besitzen. Man nennt diesen Parameter den *Empfänger* der Meldung, die zum Aufruf dieser Methode führt.

Empfänger

Der Empfänger repräsentiert das Objekt, für das die Methode aufgerufen wird und auf dessen Attribute sie zugreift. Der Typ des Empfängers bezeichnet die Klasse, zu der die Methode gehört. Die Methoden *Add* und *Clear* gehören also zur Klasse *Counter*. Obwohl sie außerhalb des Typs *Counter* deklariert sind, sind sie lokal zu *Counter*, das heißt ihr Name wird wie ein Recordfeld von *Counter* betrachtet. Durch diese Schreibweise wird – anders als bei einem abstrakten Datentyp – deutlich gemacht, welche Operationen zu einer Klasse gehören.

Benutzung von Klassen

Klassen können wie Datentypen verwendet werden. Man kann also zum Beispiel eine Variable vom Typ *Counter* deklarieren,

```
VAR c: Counter;
```

auf ihr Attribut *value* zugreifen

```
c.value
```

und ihre Methoden *Add* und *Clear* aufrufen. Methoden spricht man beim Aufruf wie Recordfelder an, also

```
c.Clear;
c.Add(5)
```

Man sagt, man schickt dem durch *c* bezeichneten Objekt die *Meldung* *Clear* oder *Add*. Dadurch drückt man aus, daß es sich nicht um einen Prozeduraufruf handelt, sondern um einen *Auftrag*, den man dem Objekt erteilt und den es zur Laufzeit interpretiert (siehe Kapitel 6).

Meldungen

Das Objekt, an das eine Meldung geschickt wird, nennt man den *Empfänger* der Meldung. Er entspricht dem formalen Empfänger-Parameter in der aufgerufenen Methode. Man versteht jetzt auch, warum der Empfänger-Parameter *vor* dem Methodennamen steht. Das ist deshalb der Fall, weil er beim Senden der Meldung ebenfalls *vor* dem Meldungsnamen steht (Abb. 4.2).

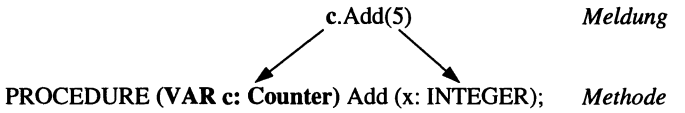


Abb. 4.2 Zuordnung von aktuellen zu formalen Parametern beim Methodenaufruf

In Kapitel 1 haben wir gesehen, daß Variablen in objektorientierten Programmen polymorph sind, also verschiedenartige Objekte enthalten können. Wie ist das aber möglich, wenn die Objekte unterschiedlich groß sind? Paßt sich etwa die Größe der Variablen der aktuellen Größe des Objekts an?

Polymorphismus ist nur dann möglich, wenn eine Variable nicht das Objekt selbst, sondern einen *Zeiger* auf das Objekt enthält. Alle Zeiger sind gleich groß, daher kann eine Variable Zeiger auf unterschiedlich große Objekte enthalten (Abb. 4.3).

Polymorphismus

Aus diesem Grund arbeitet man in der objektorientierten Programmierung häufig mit Zeigern. Klassen sind daher auch meist nicht als

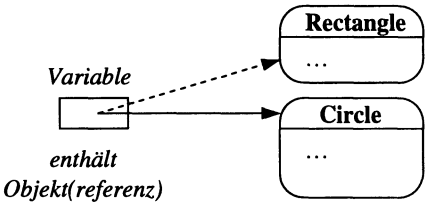


Abb. 4.3 Polymorphismus durch Objektreferenzen in Variablen

Records, sondern wie im folgenden Beispiel als Zeiger auf Records deklariert.

```
TYPE
   Counter = POINTER TO CounterDesc;
   CounterDesc = RECORD
      value: INTEGER;
   END;

PROCEDURE (c: Counter) Clear;
BEGIN
   c.value := 0
END Clear;

PROCEDURE (c: Counter) Add (x: INTEGER);
BEGIN
   c.value := c.value + 1
END Add;
```

Man beachte, daß der Empfänger-Parameter in diesem Fall ein Val-Parameter ist, während er bei Klassen, die Records sind, ein Var-Parameter sein muß. Der Grund liegt klarerweise darin, daß die Änderungen, die eine Methode an den Attributen des Empfängerobjekts vornimmt, permanent in diesem gespeichert bleiben sollen. Das erfordert bei Record-Klassen einen Var-Parameter.

Operationen als Prozedurvariablen

Im obigen Beispiel sind *Clear* und *Add* lokal zu *CounterDesc* und werden beim Aufruf wie Komponenten dieses Records angesprochen. Es stellt sich die berechtigte Frage, warum man Methoden nicht gleich als Prozedurvariablen in Records implementiert. Die Zähler-Klasse würde dann so aussehen:

```
TYPE
   Counter1 = POINTER TO Counter1Desc;
   Counter1Desc = RECORD
      value: INTEGER;
      Clear: PROCEDURE (c: Counter1);
      Add: PROCEDURE (c: Counter1; x: INTEGER);
   END;

PROCEDURE Clear (c: Counter1);
BEGIN
   c.value := 0
END;

PROCEDURE Add (c: Counter1; x: INTEGER);
BEGIN
   c.value := c.value + x
END Clear;
```

Diese Implementierung würde zu Problemen führen. Erstens müßte man, bevor man mit einem neuen Objekt arbeiten kann, seinen Prozedurvariablen die entsprechenden Prozeduren zuweisen. Das ist aufwendig und fehleranfällig. Zweitens könnte man nicht garantieren, daß *c1.Clear* und *c2.Clear* dasselbe tun. Jemand könnte in *c1* und *c2* verschiedene *Clear*-Prozeduren installiert haben. Drittens würden die installierten Prozeduren in jedem Objekt Speicherplatz belegen. Methoden belegen hingegen keinen Speicherplatz in Objekten. Viertens schließlich wäre der Aufruf geerbter und überschriebener Methoden komplizierter (siehe Kapitel 6).

Methoden sind keine Prozedur*variablen*, sondern Prozedur*konstanten*, die einer Klasse und ihren Objekten fest zugeordnet sind.

4.2 Klassen und Module

Klassen und Module sind einander sehr ähnlich: sie kapseln Daten und bieten sie über Zugriffsprozeduren an. Braucht man wirklich beide Konstrukte oder könnte man auf Module verzichten und Klassen zu Übersetzungseinheiten machen?

Diese Frage ist berechtigt. Manche Sprachen wie zum Beispiel Smalltalk bieten tatsächlich nur Klassen und keine Module an. Bei genauerer Überlegung merkt man aber, daß es vorteilhaft ist, beide Konstrukte zu haben. Sie ergänzen sich nämlich.

Subsysteme aus mehreren Klassen

In Oberon werden Klassen in Modulen deklariert. Ein Modul kann also ein *Subsystem* aus mehreren zusammengehörenden Klassen sein. Neben Klassen können Module aber auch gewöhnliche Prozeduren und Datentypen enthalten. Alles, was zur Erfüllung einer bestimmten Aufgabe benötigt wird, kann in dasselbe Modul verpackt werden.

In Sprachen wie Smalltalk oder Eiffel gibt es keine Module. Große Systeme bestehen dort einfach aus einer Vielzahl von Klassen, was nicht gerade zur Übersichtlichkeit beiträgt. Subsysteme aus mehreren zusammengehörigen Klassen wären hier sehr willkommen. In C++ wurde übrigens nachträglich das Konstrukt des *Namespace* eingeführt, das gewisse Ähnlichkeiten zu Modulen hat.

Geheimnisprinzip bei Klassen

Von Klassen verlangt man, daß sie das Geheimnisprinzip erfüllen. In Oberon-2 sind Klassen aber Records, auf deren Felder man von außen ungehindert zugreifen kann. Wie verträgt sich das?

In Oberon-2 ist nicht die Klasse ist für das Verstecken ihrer Daten zuständig, sondern das Modul, in dem sie implementiert ist. Innerhalb eines Moduls sind alle Attribute der in ihm deklarierten Klassen sichtbar, in anderen Modulen aber nur jene, die exportiert werden. Das ist vernünftig, weil ein Modul ohnehin nur zusammengehörige Klas-

sen und Prozeduren enthalten sollte. Es hat keinen Sinn, vor ihnen irgendwelche Daten zu verstecken.

Klassen oder Prozeduren, die unbeschränkt aufeinander zugreifen können sollen, werden in Oberon-2 in dasselbe Modul verpackt. Sie sind sozusagen miteinander *befreundet* und sehen voneinander mehr als Klassen anderer Module. In C++ gibt es für diesen Zweck das *Friend*-Konstrukt, mit dem Klassen und Prozeduren als befreundet bezeichnet werden können.

Module als Prozedurensammlung

Nicht alle Programme lassen sich in das Schema von Klassen und Methoden pressen. Es gibt Prozeduren (zum Beispiel numerische Funktionen), die weder von einem Zustand abhängen noch einen Zustand verändern und daher keiner Klasse zugeordnet werden können. Module erlauben es, solche Prozeduren zu einer Bibliothek zusammenzufassen; Klassen würden hingegen gekünstelt wirken.

Globale Variablen und Prozeduren

Module ermöglichen das Zusammenspiel von Klassen mit globalen Variablen und Prozeduren. In globalen Variablen können Werte gespeichert werden, die für *alle* Objekte einer Klasse gelten, ohne in jedem Objekt Speicherplatz zu beanspruchen. Globale Prozeduren erlauben es, *Operationen auf eine Klasse* auszuführen, um zum Beispiel ein neues Objekt dieser Klasse zu erzeugen und zu initialisieren. Solche Operationen können nicht als Methoden implementiert werden, denn man kann einem Objekt keine Meldung schicken, bevor es erzeugt wurde.

Klassen und Modulschnittstellen

Die Schnittstelle eines Moduls kann durch einen sogenannten *Browser* angezeigt werden (siehe Kapitel 2). Sie enthält auch die Schnittstellen der im Modul deklarierten und exportierten *Klassen*. Angenommen, die Klasse *Counter* sei folgendermaßen in einem Modul *Counters* deklariert:

```
MODULE Counters;

   TYPE
      Counter* = POINTER TO CounterDesc;
      CounterDesc* = RECORD
         value: INTEGER
      END;

   PROCEDURE (c: Counter) Clear*;
      ...
   END Clear;

   PROCEDURE (c: Counter) Add* (x: INTEGER);
      ...
   END Add;

END Counters.
```

Der Browser zeigt die Schnittstelle dieses Moduls folgendermaßen an:

```
DEFINITION Counters;

   TYPE
      Counter = POINTER TO CounterDesc;
      CounterDesc = RECORD
         PROCEDURE (c: Counter) Clear;
         PROCEDURE (c: Counter) Add (x: INTEGER);
      END;

END Counters.
```

Das Attribut *value* wurde nicht exportiert und ist daher in anderen Modulen nicht sichtbar. Die Köpfe der exportierten Methoden werden vom Browser der Übersichtlichkeit halber direkt im Record angezeigt. Dadurch wird deutlicher ausgedrückt, zu welcher Klasse die Methoden gehören.

4.3 Beispiele

Die folgenden Beispiele sollen dem Leser noch etwas mehr Sicherheit im Umgang mit Klassen vermitteln:

Klasse Set

Der Standardtyp SET erlaubt in Oberon-2 die Verwaltung von Mengen ganzer Zahlen im Bereich zwischen 0 und MAX(SET). Will man Mengen beliebig großer Zahlen haben, kann man sich dafür eine Klasse *Set* definieren:

```
DEFINITION Sets;
   TYPE
      Set = RECORD
         PROCEDURE (VAR s: Set) Init (max: INTEGER);
         PROCEDURE (VAR s: Set) Clear;
         PROCEDURE (VAR s: Set) Incl (x: INTEGER);
         PROCEDURE (VAR s: Set) Excl (x: INTEGER);
         PROCEDURE (VAR s: Set) Contains (x: INTEGER): BOOLEAN;
         PROCEDURE (VAR s: Set) Add (s1: Set);
         PROCEDURE (VAR s: Set) Subtract (s1: Set);
         PROCEDURE (VAR s: Set) Intersect (s1: Set);
      END;
END Sets.
```

Man beachte, daß *Set* ein Recordtyp ist und der Empfänger-Parameter der Methoden daher ein Var-Parameter sein muß. Die Bedeutung der Operationen ist offensichtlich, so daß man sofort ihre Implementierung angeben kann.

```
MODULE Sets;

CONST setSize = 32;  (*size of type SET*)

TYPE
   Set* = RECORD
      max-: INTEGER;  (*largest element allowed*)
      val: POINTER TO ARRAY OF SET
   END;

PROCEDURE (VAR s: Set) Init* (max: INTEGER);
BEGIN
   s.max := max; NEW(s.val, (max + setSize) DIV setSize)
END Init;

PROCEDURE (VAR s: Set) CopyTo* (VAR s1: Set);
   VAR i: INTEGER;
BEGIN
   s1.Init(s.max);
   FOR i := 0 TO s.max DIV setSize DO s1.val[i] := s.val[i] END
END CopyTo;

PROCEDURE (VAR s: Set) Clear*;
   VAR i: INTEGER;
BEGIN
   FOR i := 0 TO s.max DIV setSize DO s.val[i] := {} END
END Clear;

PROCEDURE (VAR s: Set) Incl* (x: INTEGER);
BEGIN
   IF (x > 0) & (x <= s.max) THEN
      INCL(s.val[x DIV setSize], x MOD setSize)
   END
END Incl;

PROCEDURE (VAR s: Set) Excl* (x: INTEGER);
BEGIN
   IF (x > 0) & (x <= s.max) THEN
      EXCL(s.val[x DIV setSize], x MOD setSize)
   END
END Excl;

PROCEDURE (VAR s: Set) Contains* (x: INTEGER): BOOLEAN;
BEGIN
   RETURN (x > 0) & (x <= s.max)
   & (x MOD setSize IN s.val[x DIV setSize])
END Contains;

PROCEDURE (VAR s: Set) Add* (s1: Set);
   VAR i, max: INTEGER;
BEGIN
   max := s.max; IF s1.max < max THEN max := s1.max END;
   FOR i := 0 TO max DIV setSize DO s.val[i] := s.val[i] + s1.val[i] END
```

```
END Add;

PROCEDURE (VAR s: Set) Subtract* (s1: Set);
   VAR i, max: INTEGER;
BEGIN
   max := s.max; IF s1.max < max THEN max := s1.max END;
   FOR i := 0 TO max DIV setSize DO s.val[i] := s.val[i] - s1.val[i] END
END Subtract;

PROCEDURE (VAR s: Set) Intersect* (s1: Set);
   VAR i, max: INTEGER;
BEGIN
   max := s.max; IF s1.max < max THEN max := s1.max END;
   FOR i := 0 TO max DIV setSize DO s.val[i] := s.val[i] * s1.val[i] END
END Intersect;

END Sets.
```

Die Attribut *val* der Klasse *Set* repräsentiert die eigentliche Zahlenmenge. Es wird nicht exportiert und kann von Klienten nur über Methoden verändert werden. Die Zahlenmenge ist als dynamisches SET-Array implementiert, das zur Laufzeit mit der nötigen Länge angelegt wird. Das Attribut *max* repräsentiert das größte Element, das in einem *Set*-Objekt gespeichert werden kann. Es wird hier als *read-only-Feld* exportiert. Klienten können zwar lesend darauf zugreifen, dürfen aber seinen Wert nicht verändern.

Klasse Figure

Betrachten wir als zweites Beispiel eine Klasse für Figuren in einem Grafikeditor. Hier begnügen wir uns mit der Beschreibung ihrer Schnittstelle (Modul *OS* ist in Anhang B beschrieben):

```
TYPE
   Figure = POINTER TO FigureDesc;
   FigureDesc = RECORD
      selected: BOOLEAN;
      next: Figure;
      PROCEDURE (f: Figure) Draw;
      PROCEDURE (f: Figure) Move (dx, dy: INTEGER);
      PROCEDURE (f: Figure) Select (x, y, w, h: INTEGER);
      PROCEDURE (f: Figure) Deselect;
      PROCEDURE (f: Figure) Load (VAR r: OS.Rider);
      PROCEDURE (f: Figure) Store (VAR r: OS.Rider);
   END;
```

Die Klasse *Figure* ist als Zeiger auf ein Record implementiert. Der formale Empfänger-Parameter der Methoden muß daher ein Val-Parameter sein.

4.4 Fragen

Der folgende Abschnitt gibt Antworten auf einige Fragen, die vielleicht beim Lesen dieses Kapitels aufgetaucht sind.

Können eine Methode und eine im selben Modul deklarierte Prozedur gleich heißen?

Ja. Eine Methode ist lokal zur Klasse, zu der sie gehört. Ihr Name steht daher in keinem Konflikt zu global deklarierten Namen oder zu Namen aus anderen Klassen.

Kann man eine Methode an eine Klasse binden, die in einem anderen Modul deklariert wurde?

Nein. Die Lokalität von Code und Daten ist ein wichtiges Prinzip, das die Wartung von Software erleichtert. Würde man die Methoden einer Klasse auf verschiedene Module verteilen, wäre das Lokalitätsprinzip verletzt.

Darf man einer Zeigervariablen eine Meldung schicken, wenn der formale Empfänger-Parameter ein Record ist? Also:

```
TYPE
   Ptr = POINTER TO Rec;
   Rec = RECORD ... END;

VAR p: Ptr;

PROCEDURE (VAR r: Rec) M;  ... END M;

... p.M ...   (* <-- this is valid! *)
```

Ja. Das Objekt, auf das *p* verweist, wird als Var-Parameter an *M* übergeben. Umgekehrt darf man aber einer Recordvariablen keine Meldung schicken, deren formaler Empfänger-Parameter ein Zeiger ist. Folgende Situation ist also verboten:

```
VAR r: Rec;

PROCEDURE (p: Ptr) M1; ... END M1;

... r.M1 ...   (* <-- this is invalid! *)
```

Ein Record kann nicht an einen Zeiger übergeben werden. Wenn man sowohl Variablen vom Typ *Ptr* als auch Variablen vom Typ *Rec* hat und beiden Meldungen schicken möchte, sollte man den formalen Empfänger-Parameter der Methoden als Record deklarieren.

5 Vererbung

Bis jetzt haben wir Klassen nur als abstrakte Datentypen benutzt. Das Besondere an ihnen ist aber, daß man sie erweitern kann. Die Erweiterbarkeit von Klassen ist das Neue an der objektorientierten Programmierung und der Grund dafür, daß sie konventionellen Techniken in vielen Situationen überlegen ist.

5.1 Typerweiterung

In der Praxis steht man häufig vor dem Problem, daß vorhandene Bausteine für einen gegebenen Zweck nicht genau passen. Man sucht zum Beispiel einen Zählerbaustein, der nicht nur Zahlen addieren, sondern auch Mittelwerte berechnen kann. Die vorhandene Klasse *Counter* paßt für diesen Zweck nicht genau. Was man bräuchte, wäre eine Klasse *Accumulator*, wie sie in Abb. 5.1 dargestellt ist.

Wie kann man die fehlende Funktionalität hinzufügen? Die herkömmliche Lösung besteht oft darin, den Quellcode von *Counter* zu kopieren und die zusätzliche Funktionalität in die Kopie einzubauen. Das ist aber unbefriedigend, weil man dadurch Redundanz einführt. Tritt später einmal in *Add* ein Fehler auf, so muß man ihn an *zwei* Stellen beheben. Ein Programm, das sowohl *Counter* als auch *Accumulator* verwendet, führt gewissen Code doppelt mit sich. Und schließlich hat man auch nicht immer den Quellcode des zu erweiternden Bausteins zur Verfügung.

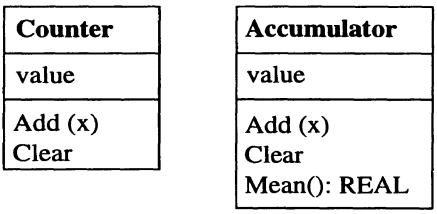


Abb. 5.1 Klasse *Accumulator* mit zusätzlicher Funktionalität

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

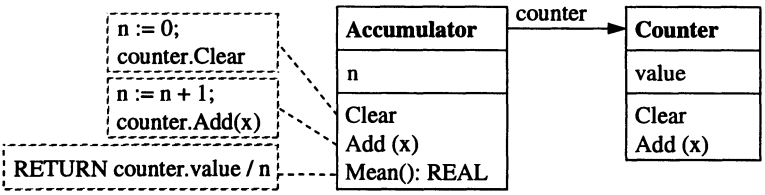


Abb. 5.2 *Counter* als Hilfsbaustein von *Accumulator*

Eine andere Lösungsmöglichkeit besteht darin, *Accumulator* auf *Counter* aufzubauen, also keinen Code zu duplizieren, sondern *Counter* als Hilfsbaustein von *Accumulator* zu benutzen (Abb. 5.2).

Der Pfeil zwischen *Accumulator* und *Counter* in Abb. 5.2 bedeutet, daß das *Accumulator*-Objekt ein *Counter*-Objekt benutzt, es also über einen Zeiger namens *counter* referenziert. Die gestrichelten Rechtecke deuten die Implementierung der Methoden von *Accumulator* an.

Bei dieser Lösung wird kein Code verdoppelt, und es ist auch nicht nötig, den Quellcode von *Counter* zu besitzen. Trotzdem sind wir mit ihr nicht ganz zufrieden. Nehmen wir an, es gibt bereits Programme, die mit *Counter*-Objekten arbeiten können. Es wäre doch schön, wenn diese Programme ohne Änderung auch mit *Accumulator*-Objekten arbeiten könnten. *Accumulator*-Objekte weisen ja alle Methoden auf, die man von *Counter*-Objekten erwartet. Trotzdem funktioniert diese Art der Wiederverwendung nicht, weil *Accumulator* und *Counter* verschiedene Typen sind.

Vererbung

Objektorientierte Sprachen bieten für dieses Problem eine bessere Lösung. Die gesuchte Klasse *Accumulator* wird aus der vorhandenen Klasse *Counter* mittels *Vererbung* (*Typerweiterung*) abgeleitet. Abb. 5.3 zeigt dies in grafischer Notation.

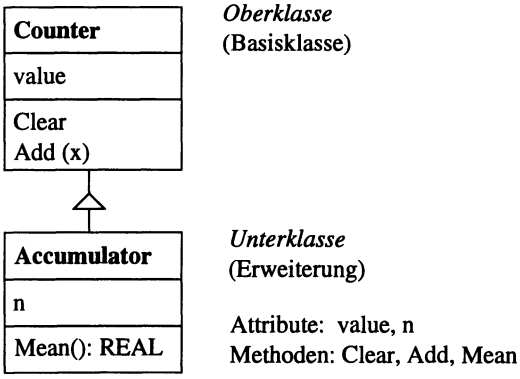


Abb. 5.3 *Accumulator* als Unterklasse von *Counter*

Vererbung bedeutet in diesem Beispiel, daß *Accumulator* alle Attribute und Methoden von *Counter* übernimmt, wie wenn sie auch in *Accumulator* deklariert worden wären. Zusätzlich kann man in *Accumulator* neue Attribute und Methoden deklarieren und dadurch die Funktionalität gegenüber *Counter* erweitern. Das Attribut *n* wurde zum Beispiel hinzugefügt, um die mit *Add* eingegebenen Werte zu zählen. Die Methode *Mean* wurde hinzugefügt, um den Mittelwert aller addierten Werte zu berechnen. *Counter* nennt man die *Oberklasse* oder *Basisklasse* von *Accumulator*. Umgekehrt ist *Accumulator* die *Unterklasse* oder *Erweiterung* von *Counter*.

Diese Lösung vermeidet ebenfalls Redundanz und hat sogar den Vorteil, daß *Accumulator* überall dort verwendet werden kann, wo bisher Counter verwendet wurde (siehe später).

Vererbung in Oberon-2

In Oberon-2 wird *Accumulator* folgendermaßen als Unterklasse von *Counter* deklariert:

```
TYPE
   Accumulator = POINTER TO AccumulatorDesc;
   AccumulatorDesc = RECORD (CounterDesc)
      n: INTEGER
   END;

PROCEDURE (a: Accumulator) Mean (): REAL;
BEGIN
   RETURN a.value / a.n
END Mean;
```

Indem man den Recordtyp *CounterDesc* in Klammern hinter das Schlüsselwort RECORD schreibt, drückt man aus, daß *CounterDesc* die Oberklasse von *AccumulatorDesc* ist. Diese Beziehung wird auch auf die dazugehörigen Zeigertypen übertragen, so daß auch *Counter* die Oberklasse von *Accumulator* ist. Bei einer Variablen vom Typ *Accumulator*

```
VAR a: Accumulator;
```

kann man nun sowohl auf die Attribute und Methoden von *Accumulator* als auch auf die von *Counter* zugreifen.

```
a.value
a.n
a.Clear
a.Add(x)
a.Mean()
```

Überschreiben von Methoden

Der Leser wird sicher schon bemerkt haben, daß es in diesem Beispiel nicht reicht, *Clear* und *Add* einfach unverändert zu erben. Beim

Löschen des Zählers mittels *Clear* sollte auch das Attribut *n* zurückgesetzt werden. Ebenso sollte beim Hinzufügen eines neuen Wertes mittels *Add* das Attribut *n* um 1 erhöht werden. Wir müssen also die geerbten Methoden anpassen. Man sagt, daß man sie in *Accumulator überschreiben* muß (*overriding*). Dazu deklariert man sie mit der gleichen Schnittstelle wie in *Counter*, bindet sie aber diesmal an *Accumulator*. Das sieht folgendermaßen aus:

```
PROCEDURE (a: Accumulator) Clear;
BEGIN
   a.n := 0;
   a.Clear^      (* Aufruf der gleichnamigen Methode aus der Oberklasse*)
END Clear;

PROCEDURE (a: Accumulator) Add (x: INTEGER);
BEGIN
   a.n := a.n + 1;
   a.Add^ (x)    (* Aufruf der gleichnamigen Methode aus der Oberklasse*)
END Add;
```

Aufruf überschriebener Methoden

Natürlich möchte man in *Clear* und *Add* nicht den Code der überschriebenen Methoden wiederholen, sondern man möchte sie an geeigneter Stelle einfach aufrufen können. Das kann man in Oberon-2 ausdrücken, indem man beim Aufruf der Methode einen Pfeil (^) hinter den Methodennamen schreibt. Der Aufruf

```
a.Clear^
```

in einer Methode der Klasse *Accumulator* bedeutet also den Aufruf der Methode *Clear* aus der Oberklasse von *Accumulator* (also aus *Counter*). Die Variable *a* muß der Empfängerparameter der rufenden Methode sein. Das folgende Beispiel zeigt, wie ein *Accumulator* benutzt werden kann:

```
VAR a: Accumulator;
...
NEW(a);
a.Clear;
a.Add(3); ... a.Add(15); ...
x := a.Mean()
```

Die Meldung *a.Clear* führt zum Aufruf der *Clear*-Methode aus *Accumulator*. Dort wird *a.n* auf 0 gesetzt, anschließend wird die *Clear*-Methode aus *Counter* aufgerufen, die *a.value* auf 0 setzt.

5.2 Klassenhierarchien

Klassenhierarchien

Aus *Counter* kann man nicht nur eine einzige Klasse sondern beliebig viele Klassen ableiten, die *Counter* auf unterschiedliche Weise erweitern. Zum Beispiel könnte man eine Unterklasse *Trigger* implementieren, die einen Zähler darstellt, der beim Erreichen eines bestimmten Wertes eine Aktion auslöst. Ebenso kann man aus einer Unterklasse wie *Accumulator* weitere Unterklassen ableiten. Zum Beispiel könnte *BoundedAccumulator* ein spezieller *Accumulator* sein, der den Zählerwert nie über einen bestimmten Maximalwert ansteigen läßt. Auf diese Weise lassen sich regelrechte Klassenhierarchien bauen wie das in Abb. 5.4 gezeigt wird.

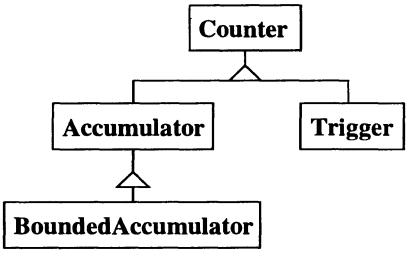


Abb. 5.4 Hierarchisches Klassendiagramm

Die Vererbungsbeziehung ist transitiv. Da *BoundedAccumulator* eine Unterklasse von *Accumulator* ist und diese wieder eine Unterklasse von *Counter*, ist *BoundedAccumulator* auch eine Unterklasse von *Counter*.

Kompatibilität

Das Besondere an der Vererbung ist nicht, daß man sich Schreibarbeit spart, weil man gewisse Dinge erben kann anstatt sie neu zu implementieren. Viel interessanter ist es, daß eine Unterklasse mit ihrer Oberklasse *kompatibel* ist und daher überall dort verwendet werden kann, wo die Oberklasse bisher schon verwendet wurde.

Vererbung stellt also eine *Ist-Beziehung* dar. Ein *Accumulator* "ist" ein *Counter*, nämlich einer, der zusätzlich zur *Counter*-Funktionalität auch Mittelwerte berechnen kann. Das sieht man besonders deutlich, wenn man die Klassen in einem Mengendiagramm darstellt (Abb. 5.5). Der Kreis mit der Beschriftung *Trigger* bezeichnet zum Beispiel die Menge aller *Trigger*-Objekte.

Man sieht aus diesem Diagramm sehr deutlich: Jeder *Accumulator* ist auch ein *Counter* (die *Accumulator*-Objekte sind eine Teilmenge der *Counter*-Objekte). Umgekehrt ist aber nicht jeder *Counter* auch ein *Accumulator*. Er könnte zum Beispiel auch ein *Trigger* sein.

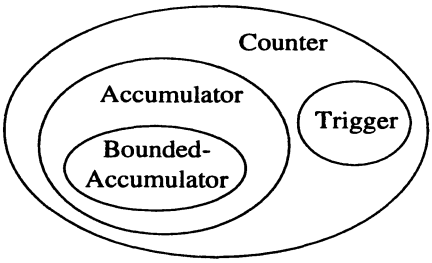


Abb. 5.5 Klassenhierarchie als Mengendiagramm

Man sieht auch, daß Typerweiterung *Spezialisierung* bedeutet. Ein *Accumulator* ist ein spezieller *Counter*, d.h. einer, der auch Mittelwerte berechnen kann. Dies führt zu der etwas paradoxen Situation, daß *Accumulator* zwar mehr Funktionalität aufweist als *Counter* (und deshalb eine *Erweiterung* ist), andererseits die Menge der *Accumulator*-Objekte aber kleiner als die Menge der *Counter*-Objekte ist (und deshalb eine *Spezialisierung* darstellt). Typerweiterung bedeutet also Spezialisierung.

5.3 Zuweisungen zwischen Objekten

Was ist eigentlich der Unterschied zwischen *Accumulator* und einer Klasse *Acc* (Abb. 5.6), die ebenfalls die Attribute *value* und *n* sowie die Methoden *Clear*, *Add* und *Mean* besitzt? Der Unterschied liegt darin, daß *Accumulator* ein *Counter* ist, *Acc* aber nicht. Alle Programme, die mit *Counter* arbeiten, können unverändert auch mit *Accumulator* arbeiten (weil das ja auch ein *Counter* ist), nicht aber mit *Acc*. *Acc* und *Counter* sind völlig verschiedene Typen.

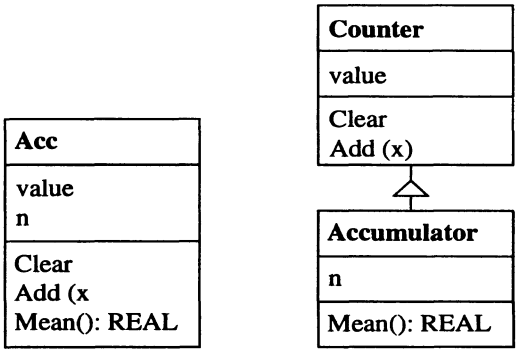


Abb. 5.6 Kompatibilität zwischen *Accumulator* und *Counter*

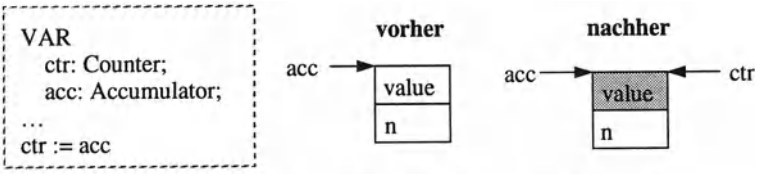


Abb. 5.7 Zuweisung eines *Accumulator*-Objekts an eine *Counter*-Variable

Weil *Accumulator*-Objekte auch *Counter*-Objekte sind, ist es möglich, ein *Accumulator*-Objekt einer *Counter*-Variablen zuzuweisen. Abb. 5.7 zeigt, was dabei geschieht.

Nach der Zuweisung *ctr := acc* zeigt *ctr* auf ein *Accumulator*-Objekt. Der Anfang dieses Objekts (der graue Teil) wird als *Counter*-Objekt interpretiert. Es sind somit folgende Zugriffe möglich

```
acc.value
acc.n
ctr.value
```

nicht jedoch

```
ctr.n
```

weil das Attribut *n* in der Klasse *Counter* nicht bekannt ist.

Statischer und dynamischer Typ

Variablen können einen statischen und einen dynamischen Typ haben. Der *statische Typ* ist derjenige Typ, mit dem eine Variable deklariert wurde. Er bestimmt, auf welche Attribute und Methoden man zugreifen darf. Der statische Typ von *ctr* ist *Counter*, daher ist *ctr.value* ein gültiger Zugriff, nicht aber *ctr.n*. Der *dynamische Typ* einer Variablen ist der Typ des Objekts, das die Variable zur Laufzeit enthält (auf das die Zeigervariable zeigt). Er wird für die Meldungsinterpretation benutzt (siehe Kapitel 6). Der dynamische Typ von *ctr* ist im obigen Beispiel *Accumulator*, weil *ctr* auf ein *Accumulator*-Objekt zeigt.

Der dynamische Typ ist eine Eigenschaft des Objekts, das in der Variablen steckt. Er wird bei der Erzeugung des Objekts festgelegt. Die Operation *NEW(acc)* erzeugt zum Beispiel ein Objekt mit dynamischem Typ *Accumulator*. Diesen dynamischen Typ behält das Objekt bei, egal welche Zeigervariable später darauf zeigt.

Der dynamische Typ einer Variablen kann eine Erweiterung ihres statischen Typs sein. Im obigen Beispiel ist der dynamische Typ von *ctr Accumulator*, also eine Erweiterung des statischen Typs *Counter*. Nun verstehen wir auch besser, was es heißt, daß Variablen in objektorientierten Programmen polymorph sein können. Sie können Objekte

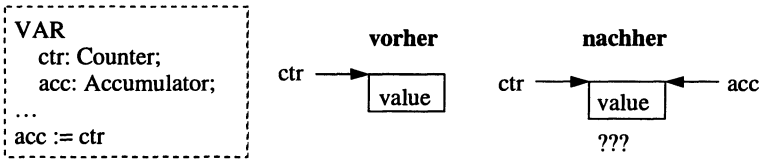


Abb. 5.8 Zuweisung eines *Counter*-Objekts an eine *Accumulator*-Variable

mit verschiedenem dynamischem Typ enthalten, solange dieser eine Erweiterung ihres statischen Typs ist. Die Variable *ctr* kann zum Beispiel Objekte des Typs *Accumulator*, *Trigger* oder *BoundedAccumulator* enthalten.

Wie sieht es mit der Zuweisung in die umgekehrte Richtung aus? Darf man einer *Accumulator*-Variablen ein *Counter*-Objekt zuweisen? Auf Grund der Ist-Beziehung ist klar, daß das nicht erlaubt ist. Nicht jeder *Counter* ist auch ein *Accumulator*, daher darf man einen *Counter* keiner *Accumulator*-Variablen zuweisen. Abb. 5.8 zeigt, was dabei passieren würde.

Nach der Zuweisung *acc := ctr* würde *acc* auf ein *Counter*-Objekt zeigen. Da ein *Counter*-Objekt aber kein Attribut *n* hat, ginge der Zugriff *acc.n* ins Leere. Daher ist diese Zuweisung verboten und wird vom Compiler als Fehler gemeldet.

Praktischer Nutzen

Was ist der praktische Nutzen dieser Zuweisungskompatibilität zwischen Unterklasse und Oberklasse? Der Nutzen besteht darin, daß jedes Programm, das schon bisher mit einem Objekt der Oberklasse arbeitete nun auch mit einem Objekt der Unterklasse arbeiten kann, ohne daß man irgendetwas am Programm ändern muß. Betrachten wir zum Beispiel die folgende Prozedur *CountData*.

```
PROCEDURE CountData (c: Counter);
   VAR x: INTEGER;
BEGIN
   LOOP
      Read(x); c.Add(x); ...
   END
END CountData;
```

Sie liest eine Folge von Zahlen und kumuliert sie im Zähler *c*, der als Parameter übergeben wird. Diese Prozedur kann man nun nicht nur als

```
CountData(ctr)
```

aufrufen, sondern auch als

```
CountData(acc)
```

Ohne es zu wissen, arbeitet die Prozedur nun mit einem *Accumulator*. Sie addiert dabei nicht nur die Zahlenreihe, sondern berechnet auch ihren Mittelwert, den man nach dem Aufruf von *CountData* als

```
m := acc.Mean()
```

abrufen kann. Dieses Beispiel zeigt die Mächtigkeit der objektorientierten Programmierung. Bestehende Programme können mit einer Vielzahl ähnlicher Datentypen arbeiten. Sie können sogar mit Datentypen arbeiten, die bei der Erstellung des Programms noch gar nicht existierten.

Record-zuweisung

In den obigen Beispielen waren die einander zugewiesenen Variablen Zeiger. Wie sieht es aus, wenn diese Variablen Records sind? Im Prinzip gilt das gleiche wie bei Zeigern. Ein Unterklassen-Objekt darf einer Oberklassen-Variablen zugewiesen werden aber nicht umgekehrt. Die Zuweisung von Records bewirkt aber eine *Projektion*, d.h. bei der Zuweisung werden Komponenten des größeren Records abgeschnitten (Abb. 5.9).

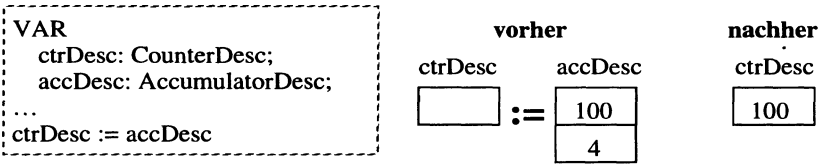


Abb. 5.9 Recordzuweisung eines *AccumulatorDesc*-Objekts an eine *CounterDesc*-Variable

Vom *AccumulatorDesc*-Objekt wird nur das Attribut *value* zugewiesen, das auch ein Attribut von *CounterDesc* ist. Nach der Zuweisung enthält *ctrDesc* also wieder (nur) ein *CounterDesc*-Objekt und kein *AccumulatorDesc*-Objekt. Der dynamische Typ von *ctrDesc* ist also auch nach der Zuweisung *CounterDesc*. Bei Recordvariablen ist der dynamische Typ *immer* gleich wie ihr statischer Typ.

Die Record-Zuweisung von Objekten ist nicht besonders nützlich. Wer will schon bei einer Zuweisung Daten verlieren? In einem bestimmten Fall genießt man allerdings beim Arbeiten mit Recordvariablen die gleichen Vorteile wie beim Arbeiten mit Zeigervariablen: wenn man sie nämlich als Var-Parameter verwendet. Wenn ein Record als Var-Parameter übergeben wird, wird nicht sein *Wert*, sondern seine *Adresse* übergeben. Da dies keine Wertzuweisung ist, wird auch nichts abgeschnitten, wie das folgende Beispiel zeigt.

```
PROCEDURE CountData (VAR c: CounterDesc);
   VAR x: INTEGER;
BEGIN
   LOOP
      Read(x); c.Add(x); ...
   END
END CountData;
```

Wird diese Prozedur als

```
CountData(accDesc)
```

aufgerufen, so gilt für den Var-Parameter *c*, daß sein dynamischer Typ *AccumulatorDesc* ist, also der Typ des aktuellen Parameters. Die Prozedur kumuliert die Zahlenwerte also in einem Accumulator.

Fassen wir zusammen:

- Eine Variable eines Recordtyps *T* kann nur Werte vom Typ *T* enthalten.
- Ein formaler Var-Parameter eines Recordtyps *T* kann Werte vom Typ *T* oder einer Erweiterung davon enthalten.
- Eine Variable eines Typs POINTER TO *T* kann Zeiger auf Werte vom Typ *T* oder einer Erweiterung davon enthalten.

Der Begriff des dynamischen Typs ist in objektorientierten Sprachen zentral und unterscheidet sie von konventionellen Sprachen. Der statische Typ ist für die strenge Typenprüfung nötig, der dynamische Typ für die Interpretation von Meldungen.

In konventionellen Sprachen mit Typenprüfung (z.B. in Pascal) gibt es nur statische Typen. In objektorientierten Sprachen ohne Typenprüfung (z.B. in Smalltalk) gibt es nur dynamische Typen: Variablen werden dort ohne Typ deklariert. In objektorientierten Sprachen mit Typenprüfung (z.B. in Oberon-2, C++, Java) gibt es sowohl statische als auch dynamische Typen.

5.4 Laufzeit-Typprüfungen

Typtest

Der dynamische Typ einer Record- oder Zeigervariablen kann in Oberon-2 zur Laufzeit mit Hilfe eines *Typtests* abgefragt werden. Der Typtest

```
IF ctr IS Accumulator THEN ...
```

liefert TRUE, wenn *ctr* vom dynamischen Typ *Accumulator* (oder einer Erweiterung davon) ist, sonst FALSE.

Typzusicherung

Wenn *ctr* vom dynamischen Typ *Accumulator* ist, sollte es eigentlich möglich sein, diese Variable wieder an *acc* zuzuweisen. Das ist auch tatsächlich möglich, wenn man für *ctr* eine sogenannte *Typzusicherung* (*type guard*) angibt. Die Zusicherung

```
ctr (Accumulator)
```

prüft zur Laufzeit, ob *ctr* vom dynamischen Typ *Accumulator* ist. Wenn das der Fall ist, wird die Variable *ctr* in diesem Bezeichner so betrachtet, wie wenn auch ihr statischer Typ *Accumulator* wäre, wie wenn sie also mit dem Typ *Accumulator* deklariert worden wäre. Wenn *ctr* nicht vom dynamischen Typ *Accumulator* ist, gibt es einen Laufzeitfehler. Die folgenden Beispiele zeigen, was nach einer Typzusicherung möglich ist.

```
acc := ctr(Accumulator)      (*zuweisungskompatibel*)
ctr(Accumulator).n           (*Attribut n ansprechbar*)
ctr(Accumulator).Mean()      (*Methode Mean ansprechbar*)
```

Die Zusicherung *ctr(Accumulator)* erfüllt eine Doppelfunktion: sie prüft, ob *ctr* vom dynamischen Typ *Accumulator* ist und wandelt den statischen Typ von *ctr* in *Accumulator* um (sie dehnt ihn auf *Accumulator* aus; siehe Abb. 5.10). Die Typzusicherung ist also eine Typumwandlung. In Oberon-2 wird allerdings zur Laufzeit geprüft, ob die Umwandlung erlaubt ist, im Gegensatz zu anderen Sprachen wie zum Beispiel C++, bei denen Typen ohne Prüfung ineinander konvertiert werden können.

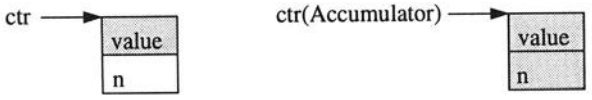


Abb. 5.10 *ctr(Accumulator)* dehnt den statischen Typ von *ctr* auf *Accumulator* aus

Wenn man den dynamischen Typ von *ctr* nicht kennt und einen Laufzeitfehler bei *ctr(Accumulator)* vermeiden möchte, stellt man der Typzusicherung einen Typtest voran:

```
IF ctr IS Accumulator THEN acc := ctr(Accumulator) END
```

Beispiele

Die folgenden Beispiele basieren auf der Klassenhierarchie in Abb. 5.4 und zeigen einige korrekte und falsche Anwendungen der Typzusicherung.

```
boundedAcc := acc (BoundedAccumulator)
```

Diese Anweisung ist korrekt, falls *acc* vom dynamischen Typ *Accumulator* oder einer Erweiterung davon ist. Wenn nicht, bewirkt die verletzte Typzusicherung einen Laufzeitfehler.

```
acc := ctr (BoundedAccumulator)
```

Für diese Anweisung gilt das gleiche wie für die vorhergegangene. Wenn *ctr* mindestens vom dynamischen Typ *BoundedAccumulator* ist, wird der Bezeichner *ctr(BoundedAccumulator)* so behandelt wie wenn er vom statischen Typ *BoundedAccumulator* wäre. Somit kann er der Variablen *acc* zugewiesen werden.

```
acc := trigger (Accumulator)
```

Diese Anweisung ist sicher falsch, weil der dynamische Typ von *trigger* nie *Accumulator* sein kann. Der Fehler wird bereits vom Compiler entdeckt.

```
ctr := trigger; acc := ctr (Accumulator)
```

Dieser Fall ist interessant. Hier wird versucht, *trigger* über *ctr* nach *acc* zu "schmuggeln". *ctr := trigger* ist korrekt; *acc := ctr (Accumulator)* ist für den Compiler zwar ebenfalls korrekt, aber die Typzusicherung bewirkt einen Laufzeitfehler, weil *ctr* vom dynamischen Typ *Trigger* und nicht *Accumulator* ist.

Hier ist noch ein Beispiel für Typtests. Wenn *ctr* vom dynamischen Typ *BoundedAccumulator* ist, liefern die folgenden drei Typtests alle TRUE.

```
ctr IS Counter
ctr IS Accumulator
ctr IS BoundedAccumulator
```

With-Anweisung

Manchmal möchte man eine Typzusicherung auf mehrere Vorkommen einer Variablen anwenden, ohne sie jedesmal hinschreiben zu müssen. Für solche Fälle gibt es in Oberon-2 die *With-Anweisung*. Statt

```
n := ctr(Accumulator).n;
mean := ctr(Accumulator).Mean()
```

kann man auch schreiben

```
WITH ctr: Accumulator DO
   n := ctr.n;
   mean := ctr.Mean()
END
```

Die Bedeutung dieser With-Anweisung ist wie folgt: wenn die Variable *ctr* vom dynamischen Typ *Accumulator* ist, wird sie in der With-Anweisung so behandelt, als ob auch ihr statischer Typ *Accumulator* wäre. Deshalb kann man auf die Felder und Methoden von *Accumulator* mit *ctr.n* und *ctr.Mean* zugreifen. Wenn *ctr* nicht vom dynamischen Typ *Accumulator* ist, gibt es einen Laufzeitfehler. Der Typtest findet nur ein einziges Mal statt, nämlich wenn die With-Anweisung betreten wird. Die With-Anweisung kann man auch auf Records anwenden, falls diese formale Var-Parameter sind, falls also überhaupt die Möglichkeit besteht, daß sich ihr dynamischer Typ von ihrem statischen Typ unterscheidet:

```
WITH ctrDesc: AccumulatorDesc DO n := ctrDesc.n; ... END
```

5.5 Beziehungen zwischen Klassen

Klassen werden meist nicht isoliert betrachtet, sondern arbeiten mit anderen Klassen zusammen, um eine bestimmte Aufgabe zu erfüllen. Man unterscheidet drei Arten von Beziehungen, die Klassen untereinander eingehen können.

Benutzt-Beziehung

Ein Objekt einer Klasse *A* kann ein Objekt einer Klasse *B benutzen*, indem es ihm Meldungen schickt oder auf seine Attribute zugreift. Dabei kann das *B*-Objekt ein Attribut von *A*, ein Parameter einer *A*-Methode oder eine globale Variable sein, die *A* kennt.

Die *Benutzt-Beziehung* wird in Klassendiagrammen durch einen Pfeil zwischen den Klassen ausgedrückt (Abb. 5.11). Die optionale Beschriftung der Pfeilenden drückt die Kardinalität der Beziehung aus. Ein *A*-Objekt steht hier mit drei *B*-Objekten in Beziehung. Ein Stern statt einer Zahl bedeutet eine Beziehung zu einer nicht näher spezifizierten Anzahl von Objekten (0, 1 oder mehrere). Falls das *B*-Objekt ein Attribut von *A* ist, wird der Anfang des Pfeils mit dem Attributnamen beschriftet (hier *b*).

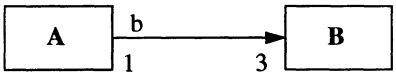


Abb. 5.11 Benutzt-Beziehung: *A* benutzt *B*

Hat-Beziehung

Ein Objekt einer Klasse *A* kann ein oder mehrere Objekte einer Klasse *B* als Bestandteile haben. Man nennt dies eine *Hat-Beziehung*. Sie ist oft schwer von einer Benutzt-Beziehung zu unterscheiden. Eine Hat-Beziehung liegt dann vor, wenn man statt "*A* hat 2 *B*-Objekte" auch sagen kann "*A* besteht aus 2 *B*-Objekten".

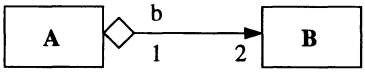


Abb. 5.12 Hat-Beziehung: A besteht aus zwei B-Objekten

Die Hat-Beziehung wird in Klassendiagrammen durch einen Pfeil mit einer Raute am Beginn ausgedrückt (Abb. 5.12). Wie bei der Benutzt-Beziehung kann man die Kardinalität und den Attributnamen als Beschriftung des Pfeils angeben. Oft wird allerdings die Hat-Beziehung grafisch nicht von der Benutzt-Beziehung unterschieden.

Ist-Beziehung

Die *Ist-Beziehung* zwischen zwei Klassen wird durch die Vererbung hergestellt. In Abb. 5.13 ist *B* eine Unterklasse von *A*. Somit *ist B* ein (Spezialfall von) *A*.

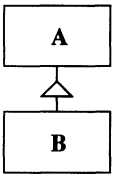


Abb. 5.13 Ist-Beziehung: Ein *B*-Objekt *ist* ein *A*-Objekt

5.6 Mehrfache Vererbung

Wenn es möglich ist, von *einer* Basisklasse zu erben, warum sollte es dann nicht auch möglich sein, von *mehreren* Basisklassen zu erben? Damit könnte man den Grad der Wiederverwendung erhöhen und die Unterklasse mit verschiedenen Basisklassen kompatibel machen. Manche objektorientierte Sprachen (z.B. C++ oder Eiffel) bieten diese Möglichkeit unter dem Namen *mehrfache Vererbung* an.

Ein häufiges Beispiel für mehrfache Vererbung ist das folgende: Ein Hausboot ist sowohl ein Haus als auch ein Boot. Man kann daher eine Klasse *Hausboot* von den beiden Klassen *Haus* und *Boot* ableiten (Abb. 5.14). Dabei erbt *Hausboot* alle Attribute und Methoden der beiden Basisklassen und ist mit beiden kompatibel, d.h. es kann wie ein Haus oder wie ein Boot verwendet werden.

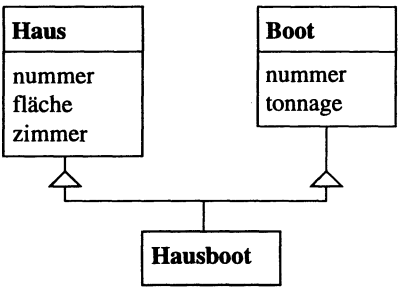


Abb. 5.14 Mehrfache Vererbung

Diese Überlegungen scheinen vernünftig zu sein. Trotzdem bietet Oberon-2 (wie viele andere objektorientierte Sprachen) keine mehrfache Vererbung an. Sie bringt nämlich einige Probleme mit sich.

Namenskonflikte

Wie man aus Abb. 5.14 sieht, können die Basisklassen gleichnamige Attribute oder Methoden enthalten. Sowohl *Haus* als auch *Boot* enthalten zum Beispiel ein Attribut *nummer*. Wenn man nun in *Hausboot* auf *nummer* zugreift, welches der geerbten Attribute spricht man dann an? Für dieses Problem gibt es verschiedene Lösungen. In Eiffel muß man zum Beispiel in *Hausboot* eines der beiden geerbten *nummer*-Attribute umbenennen. In C++ muß man bei der Benutzung von *nummer* angeben, welches der beiden geerbten Attribute man meint, indem man es mit dem Namen der entsprechenden Basisklasse qualifiziert.

Rautenstruktur

Wenn *Haus* und *Boot* selbst wiederum von einer Klasse *Besitzobjekt* abgeleitet sind, ergibt sich eine sogenannte *Rautenstruktur* (Abb. 5.15). Attribute und Methoden der Klasse *Besitzobjekt* werden sowohl von *Haus* als auch von *Boot* geerbt, und diese vererben sie wiederum an *Hausboot*. Sind sie nun in *Hausboot* einmal oder mehrmals vorhanden? In C++ wird dieses Problem durch ein eigenes Sprachkonstrukt (*virtuelle Basisklassen*) behandelt.

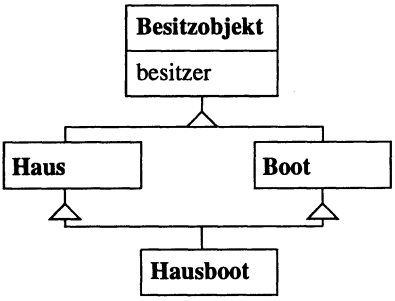


Abb. 5.15 Rautenstruktur bei mehrfacher Vererbung

Unübersichtliche Klassenhierarchien

Mehrfache Vererbung führt zu Klassenbibliotheken, die keine *Bäume* sind, sondern *gerichtete azyklische Graphen* (DAGs). Das führt zu komplizierteren Abhängigkeiten in der Klassenbibliothek und ist weniger gut durchschaubar als bei Bäumen. Daher verzichten sogar viele C++-Klassenbibliotheken auf mehrfache Vererbung, obwohl sie durch die Sprache erlaubt wäre.

Laufzeitkosten

Mehrfache Vererbung führt schließlich zu ineffizienterem Code. In C++ muß zum Beispiel bei jedem Methodenaufruf Zusatzaufwand betrieben werden, nur weil die Sprache mehrfache Vererbung unterstützt [Str89]. Dieser Zusatzaufwand ist sogar dann nötig, wenn der Programmierer von mehrfacher Vererbung gar keinen Gebrauch macht.

Aus diesen Gründen unterstützt Oberon-2 keine mehrfache Vererbung. Es gibt auch gar nicht so viele Situationen, in denen mehrfache Vererbung wirklich nötig ist. Oft kann die Ist-Beziehung zu einer der beiden Basisklassen durch eine Benutzt-Beziehung dargestellt werden. In Abb. 5.16 ist ein *Hausboot* zum Beispiel ein *Boot* und benutzt Eigenschaften von *Haus*. Natürlich geht dadurch die Kompatibilität zwischen *Hausboot* und *Haus* verloren, was aber manchmal nichts ausmacht. Sollte man wirklich nicht ohne mehrfache Vererbung auskommen, zeigt Kapitel 9.3.5, wie man sie mit einfacher Vererbung modellieren kann.

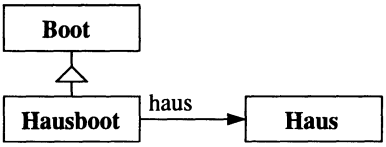


Abb. 5.16 Ersetzung einer Ist-Beziehung durch eine Benutzt-Beziehung

5.7 Fragen

Warum dürfen in einer Unterklasse nicht auch geerbte Datenfelder überschrieben werden?

Wenn es erlaubt wäre, daß ein Feld *f* in der Oberklasse *A* vom Typ INTEGER und in der Unterklasse *B* vom Typ CHAR ist, könnte der Compiler keine Typprüfungen für *a.f := a.f + 1* durchführen. Wäre *a* vom dynamischen Typ *A*, so wäre *a.f* vom Typ INTEGER und die Addition wäre legal; wäre *a* aber vom dynamischen Typ *B*, so wäre *a.f* vom Typ CHAR und die Addition wäre illegal. Die Typprüfung könnte erst zur Laufzeit erfolgen.

Kann eine Unterklasse auf Felder und Methoden ihrer Oberklasse zugreifen, wenn diese in einem anderen Modul deklariert sind und nicht exportiert werden?

Nein. In Oberon-2 sind *Module* für die Sichtbarkeit von Namen zuständig und nicht *Klassen*. Auch Unterklassen können die Modulmauer nicht überwinden.

Muß man eine exportierte Methode bei jedem Überschreiben wieder exportieren?

Eine in der Basisklasse exportierte Methode muß beim Überschreiben mit einer Exportmarke versehen werden, wenn die Unterklasse, zu der sie gehört, ebenfalls exportiert wird.

6 Dynamische Bindung

In Kapitel 5 haben wir gesehen, wie man eine Klasse erweitern und dabei Code erben kann. Die Codeersparnis ist aber nicht der Hauptvorteil der Vererbung. Viel wichtiger ist die Kompatibilität zwischen Basistyp und Erweiterung. Sie ermöglicht es, daß eine Variable zur Laufzeit Objekte verschiedenen Typs enthalten kann, die auf eine Meldung unterschiedlich reagieren.

6.1 Meldungsinterpretation

Betrachten wir nochmals die im letzten Kapitel benutzte Klassenhierarchie (Abb. 6.1). Die Basisklasse *Counter* besitzt eine *Add*-Methode, die in den Unterklassen *Accumulator* und *Trigger* überschrieben wird.

Dynamische Bindung

Eine Variable *ctr* vom statischen Typ *Counter* kann Objekte vom Typ *Counter*, *Accumulator* oder *Trigger* enthalten. Wenn wir ihr nun eine *Add*-Meldung schicken

```
ctr.Add(3)
```

so stellt sich die Frage, welches *Add* aufgerufen wird. Sinnvollerweise wird die *Add*-Methode des Objekts aufgerufen, das gerade in *ctr* ge-

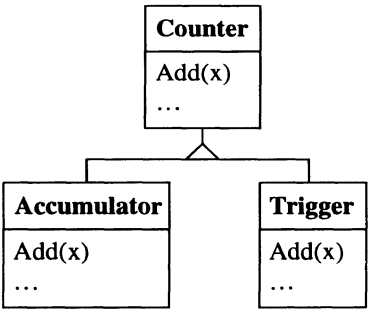


Abb. 6.1 Geerbte und überschriebene Methode *Add*

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

speichert ist. Enthält *ctr* also ein *Accumulator*-Objekt, wird die *Add*-Methode von *Accumulator* aufgerufen, enthält *ctr* ein *Trigger*-Objekt, wird die *Add*-Methode von *Trigger* aufgerufen.

Man nennt diese Art der Meldungsinterpretation *dynamische Bindung*. Meldungen werden *dynamisch* (also erst zum Zeitpunkt ihres Sendens) an eine bestimmte Methode gebunden. Die Regel lautet: *Eine Meldung obj.M bewirkt den Aufruf derjenigen Methode M, die zum dynamischen Typ von obj gehört.*

Statische Bindung

Die dynamische oder späte Bindung (*late binding*) steht im Gegensatz zur statischen oder frühen Bindung (*early binding*), die bei gewöhnlichen Prozeduraufrufen stattfindet. Bei statischer Bindung kennt der Compiler die Adresse der aufzurufenden Prozedur und kann einen direkten Aufruf erzeugen. Bei dynamischer Bindung kennt er die Adresse nicht. Sie kann erst zur Laufzeit ermittelt werden. Das Senden einer Meldung ist daher etwas langsamer als ein Prozeduraufruf. Über ein ganzes Programm verteilt, ist der Laufzeitunterschied aber kaum meßbar.

Vorteile der dynamischen Bindung

Dynamische Bindung ist das Um und Auf der objektorientierten Programmierung. Sie ermöglicht es, mit Variablen zu arbeiten, deren dynamischen Typ man nicht kennt. Will man eine Operation auf eine solche Variable anwenden, muß man sich nicht darum kümmern, von welchem dynamischen Typ sie ist. Man schickt ihr einfach eine Meldung, die sagt, was zu tun ist. Das in der Variablen gespeicherte Objekt reagiert darauf durch Aufruf der passenden Methode.

Die Prozedur *CountData* aus Kapitel 5 funktionierte eigentlich nur auf Grund der dynamischen Bindung. Rufen wir sie uns nochmals in Erinnerung.

```
PROCEDURE CountData (c: Counter);
   VAR x: INTEGER;
BEGIN
   LOOP
      Read(x); c.Add(x); ...
   END
END CountData;
```

Wenn wir diese Prozedur mit einem *Accumulator* als Parameter aufrufen

```
CountData(acc)
```

dann ist ihr formaler Parameter *c* vom dynamischen Typ *Accumulator*. Die Meldung

```
c.Add(x)
```

führt somit zum Aufruf der *Add*-Methode von *Accumulator* und addiert nicht nur *x* zum Zähler, sondern führt auch die Mittelwertberechnung durch.

CountData muß sich nicht darum kümmern, welche *Art* von Zähler sie als Parameter bekommt. Sie muß die verschiedenen Zähler nicht unterscheiden. Solange sie weiß, daß alle Zähler eine *Add*-Meldung verstehen, kann sie ihnen einfach diese Meldung schicken und darauf vertrauen, daß jeder Zähler sie richtig interpretiert.

6.2 Abstrakte Klassen

Nehmen wir an, ein Programm muß in der Lage sein, auf verschiedene Ausgabemedien wie Bildschirm, Datei oder Netzwerk zu schreiben. Es möchte diese Medien aber nicht unterscheiden, sondern sie bei allen Ausgabeoperationen gleich behandeln. Dazu müssen diese Medien im Sinne der Typerweiterung miteinander kompatibel sein. Aber welches Medium soll die Basisklasse werden und welche die Unterklassen? Eigentlich sind sie alle gleichberechtigt.

Abstrakte Klassen

Eine saubere Lösung besteht darin, das gemeinsame Verhalten aller Klassen herauszuziehen und daraus eine neue Basisklasse *Stream* zu machen, von der alle anderen Klassen abgeleitet sind (Abb. 6.2). Da es in Wirklichkeit keine Objekte der Klasse *Stream* gibt, sondern nur Objekte der Klassen *Terminal*, *File* oder *Network*, nennt man *Stream* eine *abstrakte Klasse*.

In der UML-Notation wird eine abstrakte Klasse kursiv geschrieben. Da man aber bei Benutzung von Papier und Bleistift schwer zwischen Kursivschrift und Normalschrift unterscheiden kann, setzen wir abstrakte Klassen zusätzlich in Klammern.

Abstrakte Methoden

Welche Methoden muß die Klasse *Stream* besitzen? Jedes Ausgabemedium muß Operationen wie *Write*, *WriteString* oder *WriteInt* un-

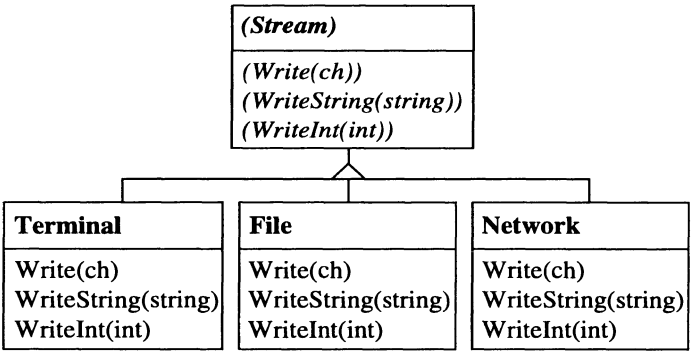


Abb. 6.2 Abstrakte Klasse *Stream* mit ihren Unterklassen

terstützen. Diese Operationen können aber in *Stream* noch nicht implementiert werden, weil sie für jedes Ausgabemedium anders aussehen. Daher gibt man in der abstrakten Klasse nur die *Schnittstelle* der Methoden an und läßt ihre Implementierung weg. Man nennt solche Operationen *abstrakte Methoden* und schreibt sie in der grafischen Notation kursiv und in Klammern.

In den Unterklassen müssen abstrakte Methoden überschrieben und implementiert werden. Wenn man dies vergißt, könnte es sein, daß eine abstrakte Methode irgendwann zur Ausführung gelangt. Das darf nicht geschehen, weil es für sie ja keine Implementierung gibt. Daher sollte man ihren Methodenrumpf nicht leer lassen, sondern zum Beipiel durch eine HALT-Anweisung implementieren, die zu einem Laufzeitfehler führt, falls die Methode aufgerufen wird. Das folgende Beispiel zeigt die Implementierung der abstrakten Klasse *Stream* und der abstrakten Methode *Write* in Oberon-2.

```
TYPE
   Stream = POINTER TO StreamDesc;        (*abstract class*)
   StreamDesc = RECORD END;

PROCEDURE (s: Stream) Write (ch: CHAR);  (*abstract method*)
BEGIN
   HALT(99)
END Write;
```

Die konkrete Unterklasse *File* wird dann zum Beispiel folgendermaßen implementiert.

```
TYPE
   File = POINTER TO FileDesc;
   FileDesc = RECORD (StreamDesc)
      fileName: ARRAY 64 OF CHAR;
      fileNo: LONGINT;
      ...
   END;

PROCEDURE (f: File) Write (ch: CHAR);
BEGIN
   ... write ch to file f...
END Write;
```

Ein Benutzerprogramm kann nun mit allen *Stream*-Varianten arbeiten, ohne sie zu unterscheiden, zum Beispiel:

```
PROCEDURE PrintErrorMsg (n: INTEGER; s: Stream);
BEGIN
   s.WriteString("--- error "); s.WriteInt(n)
END PrintErrorMsg;
```

Dem Leser stellt sich nun vielleicht folgende Frage: Wenn es für abstrakte Methoden ohnehin keine Implementierung gibt, warum muß man sie dann in der abstrakten Klasse anführen? Kann man sie dort nicht einfach weglassen und erst in den Unterklassen implementieren?

Die Antwort auf diese Frage ist klar. Man darf abstrakte Methoden nicht weglassen, weil sie die *Schnittstelle* der abstrakten Klasse festlegen, also die Operationen, die auf *Stream*-Varianten ausgeführt werden können. Würde man die abstrakte Methode *Write* in *Stream* weglassen, könnte man nicht mehr schreiben

```
stream.Write(ch)
```

weil die *Write*-Meldung in *Stream* nicht bekannt ist. Die dynamische Bindung würde nicht mehr funktionieren.

Abstrakte Klassen als Entwurfsmittel

Eine abstrakte Klasse legt die geforderte Schnittstelle für zukünftige Unterklassen fest. Sie ist also der *Entwurf* aller Klassen, die von ihr abgeleitet werden. Ein Programmierer, der später eine neue Unterklasse schreibt, weiß so bereits, welche Methoden er implementieren muß.

Während es empfehlenswert ist, in der abstrakten Klasse bereits möglichst viele *Methoden* festzulegen, ist es meist nicht sinnvoll, hier auch schon *Attribute* vorzugeben. Welche Attribute man braucht, hängt von der Implementierung der Unterklassen ab. Die Attribute der Basisklasse können in Unterklassen meist ohnehin nicht verwendet werden, ja sie behindern sogar oft die Erweiterbarkeit [WiW89].

Teilweise abstrakte Klassen

Eine abstrakte Klasse muß nicht *nur* aus abstrakten Methoden bestehen. Oft kann man bereits einige der Methoden implementieren, besonders wenn man sie auf andere abstrakte Methoden der Klasse zurückführen kann. Dies ist auch bei der Klasse *Stream* so. *WriteString* kann durch Aufrufe von *Write* implementiert werden und ist unabhängig vom verwendeten Ausgabemedium. Dasselbe gilt für *WriteInt*. Daher kann man sie schon in *Stream* implementieren. Als einzige abstrakte Methode bleibt *Write* über, die in den Unterklassen je nach Ausgabemedium anders implementiert werden muß (Abb. 6.3).

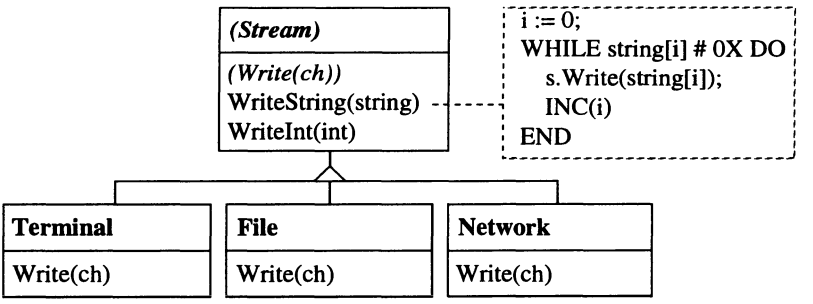


Abb. 6.3 *Stream* als teilweise abstrakte Klasse

Dies zeigt, daß es nützlich ist, möglichst viel gemeinsames Verhalten aus den Unterklassen zu extrahieren und in die (abstrakte) Basisklasse zu verlagern. Die Unterklassen werden dadurch kleiner. Neue Unterklassen sind einfacher zu implementieren, weil bereits ein großer Teil des Verhaltens von der Basisklasse geerbt werden kann.

Beispiel zum Meldungsfluß

Zum besseren Verständnis der dynamischen Bindung sehen wir uns in Abb. 6.4 den Meldungsfluß an, der auftritt, wenn man einer Variablen *stream* vom dynamischen Typ *File* die Meldung *WriteString* schickt.

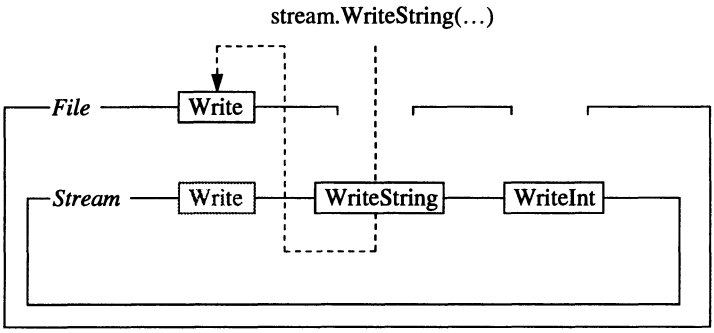


Abb. 6.4 Meldungsfluß bei *stream.WriteString(...)*

Die Unterklasse *File* kann man sich wie eine Schicht vorstellen, die um die Basisklasse *Stream* gelegt ist. Die Methode *Write* in *File* verdeckt die überschriebene Methode *Write* in *Stream*. Die Methoden *WriteString* und *WriteInt* werden in *File* nicht überschrieben. Dies wird in Abb. 6.4 so dargestellt, daß man durch die *File*-Schicht auf die geerbten *Stream*-Methoden zugreifen kann.

Wird nun eine *WriteString*-Meldung an die Variable *stream* geschickt, wird in ihrem dynamischen Typ (*File*) eine *WriteString*-Methode gesucht. *File* hat selbst keine solche Methode, erbt sie aber von *Stream*. Daher wird die *WriteString*-Methode von *Stream* aufgerufen. Diese schickt nun für jedes Zeichen des Strings eine *Write*-Meldung an das eigene Empfängerobjekt. Dieses Empfängerobjekt ist immer noch vom dynamischen Typ *File*, daher wird nach einer *Write*-Methode in der Klasse *File* gesucht, die auch gefunden und aufgerufen wird.

6.3 Fragen

Kann man eine Klasse auch einschränken anstatt sie zu erweitern, d.h. kann man in einer Unterklasse geerbte Methoden und Felder entfernen?

Nein. Würde man in einer Klasse *B* eine von *A* geerbte Methode *M* entfernen, könnte man nicht verhindern, daß ein *B*-Objekt einer *A*-Variablen *a* zugewiesen wird. Was würde *a.M* dann ergeben?

Man kann allerdings verhindern, daß eine geerbte Methode aufgerufen wird, indem man sie durch eine Methode überschreibt, die leer ist oder eine Fehlermeldung ausgibt.

Wird durch obj.M^ die Methode M aus der Oberklasse des statischen oder des dynamischen Typs von obj aufgerufen.

Es wird immer die Methode aus der Oberklasse des statischen Typs des Empfängers aufgerufen.

7 Kontrakte

Objektorientierte Programme bestehen aus kooperierenden Objekten. Jedes Objekt hat eine bestimmte Aufgabe zu erfüllen und gibt dabei anderen Objekten den Auftrag, Teilaufgaben zu bearbeiten. Um eine korrekte Zusammenarbeit zu gewährleisten muß es zwischen Auftraggeber und Auftragnehmer eine Übereinkunft geben, was bei einem Auftrag vorausgesetzt werden darf und welche Ergebisse erwartet werden. So eine Übereinkunft nennt man einen *Kontrakt* [Mey87].

Abb. 7.1 zeigt einen Kontrakt zwischen zwei Objekten *A* und *B*. *A* gibt *B* den Auftrag *M* (schickt ihm die Meldung *M*). Der Kontrakt besteht aus Garantien und Erwartungen. *A* garantiert, daß vor dem Auftrag gewisse Voraussetzungen gelten, die natürlich den Erwartungen von *B* entsprechen müssen. Dafür garantiert *B* gewisse Ergebnisse, die wiederum den Erwartungen von *A* entsprechen müssen.

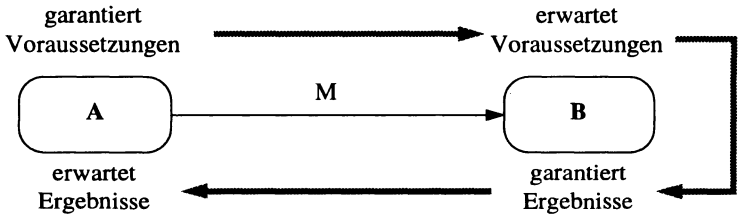


Abb. 7.1 Kontrakt zwischen *A* und *B* für den Auftrag *M*

B kann zum Beispiel erwarten, daß ein Eingangsparameter *x* eine positive und keine negative Zahl ist, dafür garantiert *B*, daß der Rückgabeparameter die Wurzel von *x* ist. *A* muß sicherstellen, daß *x* positiv ist und darf sich dafür darauf verlassen, daß er die Wurzel von *x* zurückbekommt.

Vor- und Nachbedingungen

Garantien und Erwartungen lassen sich durch *Vorbedingungen* (*preconditions*) und *Nachbedingungen* (*postconditions*) einer Methode ausdrücken. Die Notation

```
{pre} M {post}
```


H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

bedeutet: Wenn vor Ausführung von *M* die Vorbedingung *pre* gilt, dann garantiert *M*, daß nach seiner Ausführung die Nachbedingung *post* gilt. Wenn *pre* nicht gilt, ist *M* nicht an den Kontrakt gebunden und braucht auch *post* nicht zu garantieren.

ASSERT

Die Bedingungen *pre* und *post* sind Boolesche Ausdrücke (*Assertionen*) und lassen sich in Oberon-2 durch die Standardprozedur ASSERT ausdrücken.

```
PROCEDURE (...) M (x: INTEGER; VAR y: INTEGER);
BEGIN
   ASSERT(x >= 0);
   ...
   ASSERT((y >= 0) & (y < x))
END M;
```

ASSERT(b) prüft zur Laufzeit, ob die Bedingung *b* erfüllt ist. Falls sie nicht erfüllt ist, wird das Programm mit einem Laufzeitfehler abgebrochen. Assertionen sind wichtige Hilfsmittel, um Programmfehler frühzeitig zu erkennen. Sobald eine Erwartung oder eine Garantie nicht erfüllt ist, wird der Fehler gemeldet, so daß nicht mit falschen Annahmen weitergerechnet wird.

Wenn ein Programm genügend getestet ist und man Vertrauen hat, daß alle Vor- und Nachbedingungen immer erfüllt sind, kann man die in ASSERT vorgenommenen Prüfungen ausschalten. Durch eine Compileroption kann man angeben, daß kein Code für ASSERT erzeugt werden soll. Das Programm ist dann schneller. Aus Sicherheitsgründen sollte man allerdings die Assertionen auch in ausgetesteten Programmen belassen. Sie kosten wenig Zeit und erhöhen die Zuverlässigkeit des Programms.

7.1 Klassenassertionen

Die Korrektheit einer Klasse kann durch *Klassenassertionen* geprüft werden. Dazu muß man für jede Methode eine Vor- und eine Nachbedingung sowie für die gesamte Klasse eine Klasseninvariante angeben.

Vorbedingung

Die *Vorbedingung* einer Methode beschreibt die Voraussetzungen für ihren korrekten Ablauf und ist ein Boolescher Ausdruck, in dem Eingangsparameter der Methode und Attribute der Klasse vorkommen können.

Nachbedingung

Die *Nachbedingung* einer Methode beschreibt, was die Methode dem Rufer garantiert (unter der Voraussetzung, daß die Vorbedingung erfüllt war). Sie ist ein Boolescher Ausdruck, in dem Eingangs- und Ausgangsparameter der Methode sowie Attribute der Klasse vorkommen können.

Eine *Klasseninvariante* ist ein Boolescher Ausdruck, in dem die Attribute der Klasse vorkommen können. Sie beschreibt die Bedingung, die vor und nach dem Aufruf jeder Methode gelten muß. Diese Bedingung ist invariant, darf also durch eine Methode nicht verletzt werden (während der Ausführung der Methode kann sie allerdings vorübergehend außer Kraft gesetzt werden).

Klasseninvariante

Das Beispiel in Abb. 7.2 zeigt die Klassenassertionen einer Klasse Text, die einen Textpuffer *buf* der Länge *len* verwaltet und eine Methode *Insert* hat, mit der eine Zeichenkette *s* an der Position *pos* in den Textpuffer eingefügt werden kann. *len'* und *buf'* bedeuten dabei die Werte der Attribute *len* und *pos* nach Ausführung der Methode *Insert*.

Text	
buf len	
Insert (pos, s) ...	

inv len >= 0

Insert

pre 0 <= pos <= len

post (len' = len + Len(s)) & (buf'[pos .. pos+Len(s)] = s)

Abb. 7.2 Klassenassertionen einer Klasse *Text*

Mit Klassenassertionen lassen sich die Korrektheitsbedingungen einer Klasse formulieren. Eine Klasse ist korrekt, wenn folgende Bedingungen gelten:

Korrektheit einer Klasse

- Nach der Initialisierung jedes Objekts der Klasse gilt *inv*

  {TRUE} Initialisierung {inv}

- Jede Methode *M* hält *inv* aufrecht und ist bezüglich *pre* und *post* korrekt

  {inv & pre} M {inv & post}

7.2 Subkontrakte

Wenn man eine Methode aufruft, die in einer Unterklasse überschrieben wird, dann wird wegen der dynamischen Bindung die Unterklassenmethode aufgerufen. Die Oberklasse ist nun nicht mehr alleine für die Erfüllung ihres Kontrakts zuständig, sondern bedient sich der Unterklasse als *Subkontraktor*.

Wie immer, wenn ein Auftrag an einen Subkontraktor vergeben wird, darf der ursprüngliche Kontrakt nicht verletzt werden, das heißt, der Subkontraktor darf *nicht mehr erwarten* als der Kontraktor und darf auch *nicht weniger leisten*. Abb. 7.3 zeigt ein Beispiel für einen gültigen Subkontrakt.

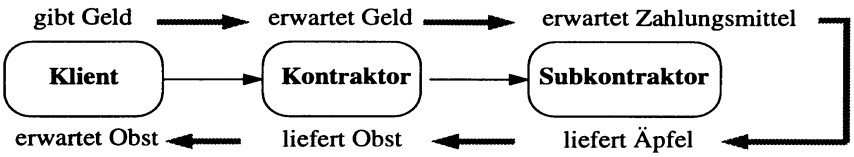


Abb. 7.3 Gültiger Subkontrakt

Der ursprüngliche Kontrakt lautet in diesem Beispiel: Wenn der Klient dem Kontraktor Geld gibt, liefert ihm dieser Obst. Der Subkontraktor könnte nun einfach ebenfalls Geld erwarten und Obst liefern. Dann wäre der Kontrakt unverändert und daher per definitionem gültig.

Der Subkontraktor darf den Kontrakt aber auch *verändern*, solange er sich an gewisse Spielregeln hält. Er darf nicht mehr erwarten und nicht weniger garantieren als der Kontraktor. In Abb. 7.3 erwartet der Subkontraktor nicht unbedingt Geld, sondern ist auch mit jedem anderen Zahlungsmittel (z.B. einem Scheck) einverstanden. Seine Erwartung ist also *allgemeiner* (d.h. weniger strikt) als die des Kontraktors und daher gültig. Umgekehrt liefert der Subkontraktor Äpfel. Da der Klient Obst erwartet und Äpfel Obst sind, wird der Klient zufrieden sein. Der Subkontraktor darf also etwas *Spezielleres* liefern, solange dies die Garantien des Kontraktors nicht verletzt.

Das Beispiel in Abb. 7.4 zeigt hingegen einen ungültigen Subkontrakt. Der Subkontraktor erwartet Geldscheine. Wenn ihm der Klient beliebiges Geld gibt (also z.B. Münzen), ist er nicht zufrieden. Der Subkontraktor erwartet mehr als der ursprüngliche Kontraktor, was nicht erlaubt ist. Umgekehrt garantiert der Subkontraktor weniger als der Kontraktor, denn er liefert etwas Allgemeineres als Obst, nämlich Lebensmittel. Damit ist der Klient unter Umständen nicht zufrieden. Auch hier ist also der Kontrakt verletzt.

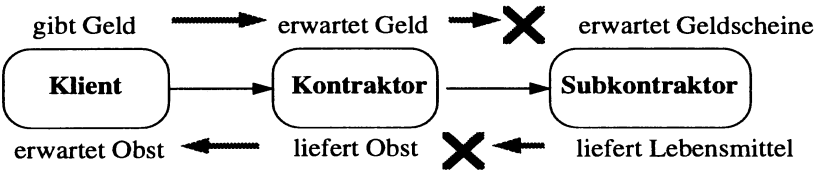


Abb. 7.4 Ungültiger Subkontrakt

Formal ausgedrückt müssen folgende Bedingungen gelten, damit ein Subkontrakt zwischen Oberklasse und Unterklasse gültig ist.

- Die *Vorbedingung* einer Unterklassen-Methode (pre_{sub}) darf *schwächer* sein als die Vorbedingung der entsprechenden Oberklassen-Methode (pre_{base}). Das entspricht der Implikation

  $$pre_{base} \Rightarrow pre_{sub}$$

- Die *Nachbedingung* einer Unterklassen-Methode ($post_{sub}$) darf *stärker* sein als die Nachbedingung der entsprechenden Oberklassen-Methode ($post_{base}$). Das entspricht der Implikation

  $$post_{sub} \Rightarrow post_{base}$$

- Es muß sowohl die Klasseninvariante der Oberklasse als auch die der Unterklasse gelten.

  $$inv_{base} \ \& \ inv_{sub}$$

Diese Aussagen kann man nochmals grafisch veranschaulichen. Die Dicke der Pfeile in Abb. 7.5 stellt die Menge von Zuständen dar, die eine bestimmte Vor- oder Nachbedingung erfüllen. Die Unterklasse darf weniger erwarten (d.h. mehr Anfangszustände erlauben, daher dickerer Pfeil) und mehr garantieren (d.h. weniger Endzustände erlauben, daher dünnerer Pfeil) als die Oberklasse. Wenn das erfüllt ist, ist jeder Zustand von pre_{base} auch in pre_{sub} und jeder Zustand von $post_{sub}$ auch in $post_{base}$.

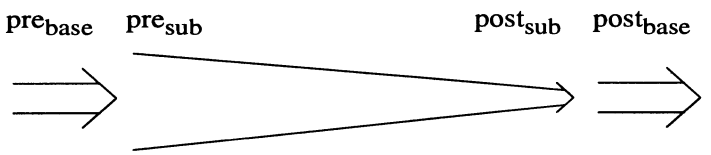
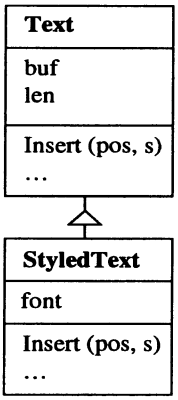


Abb. 7.5 Erlaubte Änderung von Vor- und Nachbedingung einer Methode

Nun wollen wir diese Regeln auf das Beispiel mit der Textklasse anwenden. Angenommen, jemand möchte aus *Text* eine Unterklasse *StyledText* ableiten, die zusätzlich zum Text Schriftarten (Fonts) verwaltet. Wenn neuer Text mit *Insert(pos, s)* eingefügt wird, muß auch die Font-Information von *s* eingestellt werden. Wir müssen also *Insert* überschreiben. Da wir aber seine Schnittstelle nicht verändern dürfen, legen wir folgende Regeln fest:

- Wenn *pos* > 0 ist, wird der Font von *s* aus dem Font des Vorgängerzeichens *buf[pos-1]* übernommen.
- Wenn *pos* ≤ 0 ist, wird für *s* der Standardfont verwendet. Die Einfügeposition ist in diesem Fall *|pos|*.

Der Einfachheit halber nehmen wir an, daß die Schriftarten in einem Array *font* gespeichert sind, so daß *font[i]* die Schriftart des Zeichens *buf[i]* enthält. Abb. 7.6 zeigt die Klasseninvarianten von *StyledText*.




```
inv     (len >= 0) &
        (font[0..len-1] is font of buf[0..len-1])

Insert
pre     -len <= pos <= len
post    (len' = len+Len(s)) &
        (buf'[|pos|..|pos|+Len(s)] = s) &
        (  (pos  0) & (font[|pos|..|pos|+Len(s)] = default font)
        |  (pos>0) & (font[pos..pos+Len(s)] = font[pos-1])
        )
```

Abb. 7.6 Klasseninvarianten der Unterklasse *StyledText*

Stellt die Unterklasse *StyledText* mit ihren Klasseninvarianten nun einen gültigen Subkontrakt zu *Text* dar oder nicht? Das läßt sich leicht prüfen. Wie man aus Abb. 7.2 und Abb. 7.6 sieht, gilt sowohl $pre_{base} \Rightarrow pre_{sub}$

$$0 \leq pos \leq len \quad \Rightarrow \quad -len \leq pos \leq len$$

als auch $post_{sub} \Rightarrow post_{base}$

```
(len' = len+Len(s)) & (buf'[|pos| .. |pos|+Len(s)] = s) &
(  (pos≤0) & (font[|pos|..|pos|+Len(s)] = default font)
|  (pos>0) & (font[pos..pos+Len(s)] = font[pos-1])
)
   ⇒
(len' = len+Len(s)) & (buf'[|pos| .. |pos|+Len(s)] = s)
```

StyledText ist also ein legaler Subkontraktor von *Text*. Das heißt, daß jeder Klient, der bisher mit *Text* arbeitete, nun auch mit *StyledText* arbeiten kann, ohne daß dadurch sein bisheriger Kontrakt verletzt wird.

7.3 Parameter überschriebener Methoden

In Kapitel 5 wurde festgelegt, daß beim Überschreiben einer Methode die Parametertypen nicht verändert werden dürfen. Parametertypen drücken aber Vor- und Nachbedingungen aus. Daher sollte man sie eigentlich in einer überschreibenden Methode abschwächen oder verstärken dürfen, wie dies in Kapitel 7.2 erklärt wurde.

Betrachten wir Abb. 7.7. Die Methode *M* der Klasse *A* hat einen Eingangsparameter vom Typ *T* und gibt auch ein Resultat vom Typ *T* zurück. In der Unterklasse *B* wird *M* überschrieben. Wie darf man seine Parametertypen ändern, ohne den Kontrakt zwischen *A* und seinen Klienten zu verletzen?

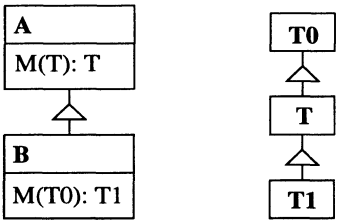


Abb. 7.7 Beispiel für geänderte Parametertypen in Unterklassen-Methoden

Nach den Regeln aus Kapitel 7.2 darf die Klasse *B* weniger erwarten und mehr garantieren als *A*. Das bedeutet im einzelnen:

- Ein *Eingangsparameter* darf in einer überschreibenden Methode von einem *allgemeineren* Typ sein (d.h. von einem Basistyp). Statt *T* darf also *T0* verwendet werden. Da jedes *T*-Objekt auch ein *T0*-Objekt ist, können bisherige Klienten ruhig weiter *T*-Objekte übergeben. Es gilt dann *Kontravarianz*

  $$pre_{base} \Rightarrow pre_{sub} \qquad T \Rightarrow T0$$

  Diese Änderung eines Parametertyps nennt man *Kontravarianz*, weil sich der Parametertyp *entgegen* dem Empfängertyp ändert (der Empfängertyp wird spezieller, der Parametertyp allgemeiner). Kontravarianz ist bei Eingangsparametern typsicher, aber nicht besonders nützlich.

- Ein reiner *Ausgangsparameter* (d.h. ein Funktionswert) darf in der überschreibenden Methode von einem *spezielleren* Typ sein (d.h. von einem Untertyp). Statt *T* darf also *T1* verwendet werden. Da *Kovarianz*

T1-Objekte auch *T*-Objekte sind, kann die Methode ruhig ein *T1*-Objekt an einen Klienten zurückgeben, der ein *T*-Objekt erwartet. Es gilt dann

$$post_{sub} \Rightarrow post_{base} \qquad T1 \Rightarrow T$$

Diese Änderung des Parametertyps nennt man *Kovarianz*, weil sich der Parametertyp in *dieselbe* Richtung ändert wie der Empfängertyp (beide werden spezieller). Kovarianz ist bei reinen Ausgangsparametern typsicher.

- Ein Übergangsparameter *(VAR x: T)* darf sich in einer überschreibenden Methode nicht ändern. Da er sowohl ein Eingangs- als auch ein Ausgangsparameter ist, muß gelten

  $$(T \Rightarrow newType) \ \& \ (newType \Rightarrow T)$$

  woraus folgt, daß *newType* = *T* sein muß.

In Oberon-2 (und in den meisten anderen objektorientierten Sprachen) wird Kontravarianz von Eingabeparametern nicht unterstützt. Kovarianz von Funktionstypen wird hingegen in einigen (nicht in allen) Implementierung von Oberon-2 unterstützt, weil dies nützlich und typsicher ist. Die Typen von Var-Parametern schließlich müssen beim Überschreiben von Methoden unverändert bleiben.

8 Typische Anwendungen

Objektorientierte Programmierung führt in einigen Situationen zu sehr eleganten Lösungen, in anderen bringt sie jedoch wenig, ja sie kann sogar – wenn man sie falsch anwendet – zusätzliche Komplexität verursachen. Im wesentlichen sind es die folgenden Anwendungen, für die sich Klassen besonders lohnen:

- Abstrakte Datentypen
- Generische Bausteine
- Heterogene Datenstrukturen
- Austauschbares Verhalten
- Erweiterung bestehender Bausteine
- Halbfabrikate

Wann immer man generische Bausteine, heterogene Datenstrukturen oder austauschbares Verhalten benötigt, bieten sich Klassen als Implementierungstechnik an. Der erfahrene Programmierer hat ein Auge für solche Situationen und weiß, wie er sie mit Klassen nutzbringend lösen kann.

8.1 Abstrakte Datentypen

Klassen als Strukturierungsmittel

Klassen sind ein ausgezeichnetes *Strukturierungsmittel*. Sie gruppieren zusammengehörende Daten und Operationen und sorgen damit für Ordnung in Programmen. Sie helfen, unwichtige Details vor Klienten zu verbergen und reduzieren damit die Komplexität von Software.

Selbst wenn man Vererbung und dynamische Bindung gar nicht benutzt, kann es sinnvoll sein, einen Datentyp als Klasse zu implementieren, um ihn zu einem identifizierbaren, abgeschlossenen Baustein zu machen. Nehmen wir als Beispiel eine Klasse zur Ansteuerung einer seriellen RS232-Schnittstelle. Details, wie Steuerregister, Handshake-Protokoll und Signale, können hinter folgender Schnittstelle verborgen werden:

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

```
TYPE
   RS232 = RECORD
      PROCEDURE (VAR x: RS232) Init
         (address, bitRate, dataBits, stopBits, parity: LONGINT);
      PROCEDURE (VAR x: RS232) Send (ch: CHAR);
      PROCEDURE (VAR x: RS232) Receive (VAR ch: CHAR);
   END;
```

Diese Schnittstelle ist einfach, hardwareunabhängig und gegenüber Änderungen robust. Man sollte allerdings überlegen, ob man den Baustein wirklich als *Typ* braucht. Wenn nicht, ist ein *Modul* wie das folgende das einfachere und effizientere Konstrukt:

```
DEFINITION RS232;
   PROCEDURE Init (bitRate, dataBits, stopBits, parity: INTEGER);
   PROCEDURE Send (ch: CHAR);
   PROCEDURE Receive (VAR ch: CHAR);
END RS232.
```

Kosten der Datenabstraktion

Datenabstraktion ist nicht gratis. Eine Klasse beseitigt nicht nur Komplexität, sondern führt auch ein gewisses Maß an neuer Komplexität ein. Schließlich wird ein neuer Baustein mit Operationen definiert, von denen man sich merken muß, wie sie heißen, welche Parameter sie haben und was sie tun. Die durch die Datenabstraktion gewonnene Vereinfachung muß groß genug sein, um die neu eingeführte Komplexität wettzumachen. Es ist zum Beispiel nicht sinnvoll, für das Gehalt einer Person folgende Klasse einzuführen:

```
TYPE
   Salary = RECORD
      amount: INTEGER;
      PROCEDURE (s: Salary) Set (value: INTEGER);
      PROCEDURE (s: Salary) Get (VAR value: INTEGER);
      PROCEDURE (s: Salary) Increment (value: INTEGER);
   END;
```

Die Klasse *Salary* führt mehr Komplexität ein, als sie beseitigt. An ihrer Stelle wäre der Standardtyp INTEGER völlig ausreichend. Das Beispiel mag überspitzt sein, aber Fehler dieser Art passieren häufig, vor allem dann, wenn man das Gefühl hat, *alles* mit Klassen ausdrükken zu müssen.

Der Einsatz von Klassen zur Datenabstraktion ist zwar nicht besonders neu, er ist aber diejenige Anwendung der objektorientierten Programmierung, die sich am häufigsten anbietet. Vererbung und dynamische Bindung sind nur in wenigen Programmen nützlich, Datenabstraktion hingegen in fast allen.

Datenabstraktion und die durch sie gewonnenen Strukturierungsmöglichkeiten sind ein wesentlicher Grund für die Popularität objektorientierter Sprachen. In Modula-2 oder Ada ist Datenabstraktion eine längst vertraute Technik. Für Cobol- oder C-Programmierer bedeutet sie jedoch einen gewaltigen Fortschritt. So ist es auch zu erklären, daß manche die objektorientierte Programmierung als revolutionär betrachten, während andere sie weniger spektakulär finden.

8.2 Generische Bausteine

Ein Baustein heißt *generisch*, wenn er mit verschiedenen Arten von Objekten arbeiten kann. Sprachen, wie Ada, Eiffel oder C++ bieten Generizität als eigenes Sprachkonstrukt an. Man kann Generizität aber auch durch Vererbung simulieren.

Generische Binärbäume

Betrachten wir ein einfaches Beispiel: einen generischen Binärbaum. Die Algorithmen zum Einfügen und Suchen von Objekten im Baum sind unabhängig davon, ob die Objekte Zahlen, Zeichenketten oder komplexere Daten sind. Es liegt daher nahe, die Algorithmen so zu implementieren, daß sie nicht auf eine *bestimmte* Objektart ausgelegt sind, sondern mit *verallgemeinerten* Objekten arbeiten, die später durch Zahlen oder Zeichenketten ersetzt werden können. Ein Binärbaum dieser Art könnte wie in Abb. 8.1 aussehen.

Anstatt mit Zahlen oder Zeichenketten arbeitet dieser Baum mit Knoten vom abstrakten Typ *Node*. Obwohl die Struktur der Knoten unbekannt ist, muß der Baum gewisse Annahmen über sie machen: Knoten sollen einen linken und einen rechten Sohn haben und es muß möglich sein, sie miteinander zu vergleichen, um sie im Baum zu suchen. Diese Annahmen werden durch die Schnittstelle der abstrakten Klasse *Node* ausgedrückt. Diese Annahmen reichen aus, um die Operationen des Binärbaums zu implementieren.

```
TYPE
   Tree = RECORD
      root: Node
   END;
```

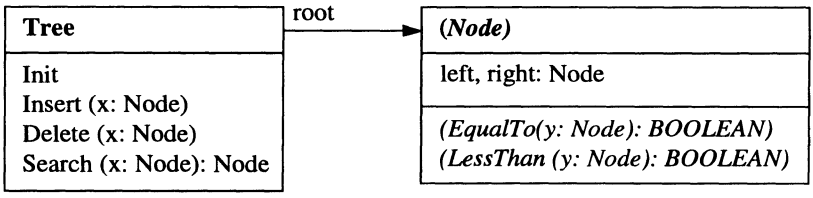


Abb. 8.1 Generische Klasse *Tree* mit abstrakter Knotenklasse

```
PROCEDURE (VAR t: Tree) Init;
BEGIN
    t.root := NIL
END Init;

PROCEDURE (VAR t: Tree) Insert (x: Node);
    VAR this, father: Node;
BEGIN
    this := t.root; x.left := NIL; x.right := NIL;
    WHILE this # NIL DO
        father := this;
        IF x.EqualTo(this) THEN RETURN (*don't insert duplicates*) END;
        IF x.LessThan(this) THEN this := this.left ELSE this := this.right END
    END;
    IF t.root = NIL THEN t.root := x
    ELSIF x.LessThan(father) THEN father.left := x
    ELSE father.right := x
    END
END Insert;

PROCEDURE (VAR t: Tree) Search (x: Node): Node;
    VAR this: Node;
BEGIN
    this := t.root;
    WHILE (this # NIL) & ~ x.EqualTo(this) DO
        IF x.LessThan(this) THEN this := this.left ELSE this := this.right END
    END;
    RETURN this
END Search;

PROCEDURE (VAR t: Tree) Delete (x: Node);
    VAR this, father, p, q: Node;
BEGIN
    this := t.root;
    WHILE (this # NIL) & ~ x.EqualTo(this) DO
        father := this;
        IF x.LessThan(this) THEN this := this.left ELSE this := this.right END
    END;
    IF this # NIL THEN (*x.EqualTo(this); find a node p that can replace this*)
        IF this.right = NIL THEN p := this.left
        ELSIF this.right.left = NIL THEN p := this.right; p.left := this.left
        ELSE (*p := smallest node greater than this*)
            p := this.right; WHILE p.left # NIL DO q := p; p := p.left END;
            q.left := p.right; p.left := this.left; p.right := this.right
        END;
        IF this = t.root THEN t.root := p
        ELSIF this.LessThan(father) THEN father.left := p
        ELSE father.right := p
        END
    END
END Delete;
```

Will man nun in diesem Binärbaum Zeichenketten oder Zahlen speichern, muß man diese mit *Node* kompatibel machen (Abb. 8.2). Man implementiert sie als Unterklassen von *Node* und kann sie dann daher in den Methoden von *Tree* anstelle von *Node* verwenden.

Speicherung von Zeichenketten im Binärbaum

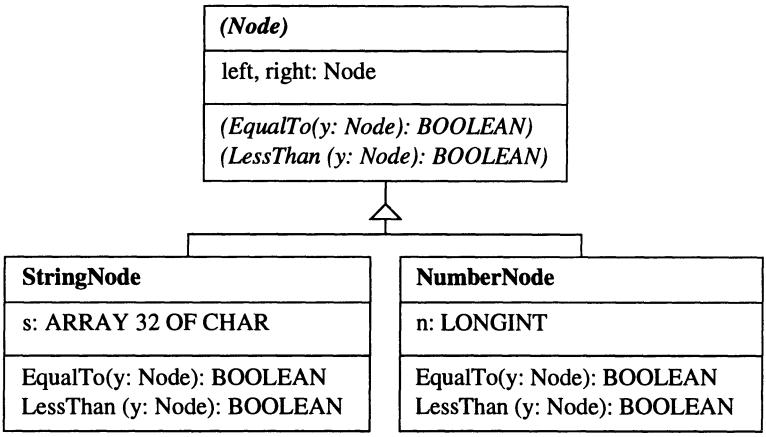


Abb. 8.2 Konkrete *Node*-Klassen

Die Implementierung von *StringNode* sieht in Oberon-2 folgendermaßen aus:

```
TYPE
   StringNode = POINTER TO StringNodeDesc;
   StringNodeDesc = RECORD (NodeDesc)
      s: ARRAY 32 OF CHAR
   END;

PROCEDURE (x: StringNode) EqualTo (y: Node): BOOLEAN;
BEGIN
   RETURN x.s = y(StringNode).s
END EqualTo;

PROCEDURE (x: StringNode) LessThan (y: Node): BOOLEAN;
BEGIN
   RETURN x.s < y(StringNode).s
END LessThan;
```

Man beachte, daß der Parameter *y* vom statischen Typ *Node* ist, weil überschriebene Methoden die gleichen Parametertypen haben müssen wie die entsprechenden Methoden der Basisklasse. Es ist daher eine Typzusicherung *y(StringNode)* nötig, um auf das Feld *y.s* zugreifen zu können.

Das folgende Beispiel zeigt, wie Zeichenketten in einen Baum vom Typ *Tree* eingefügt werden.

```
VAR t: Tree; x: StringNode;
...
NEW(x); x.s := ...; t.Insert(x);
```

Die Methode *Insert* schickt dabei den Baumknoten die Meldungen *EqualTo* und *LessThan*, die dynamisch an die entsprechenden Methoden von *StringNode* gebunden werden. Auf die gleiche Weise verfährt man, wenn man Zahlen im Baum speichern will: man leitet aus *Node* eine Klasse *NumberNode* ab und überschreibt die Methoden *EqualTo* und *LessThan*.

Mit dem generischen Typ *Tree* haben wir folgendes erreicht:

- *Tree* kann mit allen Objekten arbeiten, deren Klasse von *Node* abgeleitet ist, und die sich mittels *EqualTo* und *LessThan* vergleichen lassen.
- *Node* dient als Muster für zukünftige Knotenklassen.
- *Tree* kann wiederverwendet werden, ohne geändert oder auch nur neu compiliert zu werden.

Generizität als eigenes Sprachkonstrukt

Manche Sprachen wie C++ oder Eiffel bieten Generizität als eigenes Sprachkonstrukt an. In Eiffel kann eine Klasse mit einem Typ *T* parametrisiert werden, der in eckigen Klammern auf den Klassennamen folgt. Ein generischer Kellerspeicher (Stack) sieht zum Beispiel in Eiffel folgendermaßen aus:

```
class Stack [T]
    ...
    Push (x: T) is do ... end;
    Pop: T is do ... end;
end
```

Die Operationen *Push* und *Pop* arbeiten mit Objekten vom Typ *T*. Bei der Deklaration einer *Stack*-Variablen kann man *T* durch einen konkreten Typ wie INTEGER ersetzen und erhält dadurch einen Stack, der mit ganzen Zahlen arbeiten kann:

```
intStack: Stack[INTEGER];
i: INTEGER;
...
intStack.Push(3); ...
 i := intStack.Pop; ...
```

Man kann *Stack* für beliebige Elementtypen verwenden, ohne zuerst eine neue Elementklasse (wie z.B. *IntegerNode*) einführen zu müssen. Generizität dieser Art ist aber nur für die allereinfachsten Bausteine

geeignet, die keinerlei Annahmen über die von ihnen verwalteten Elemente machen, zum Beispiel Stacks, Queues oder unsortierte Listen. Die meisten nützlichen Bausteine wie Bäume, Mengen oder sortierte Listen setzen von ihren Elementen zumindest voraus, daß man sie vergleichen kann. Eiffel erlaubt daher, daß man den abstrakten Typ *T* näher spezifiziert. Die Klassendeklaration

```
class Tree [T -> Node]
    ...
end
```

spezifiziert, daß der konkrete Typ, mit dem *Tree* später parametrisiert wird, *Node* oder eine Erweiterung davon sein muß. Hier arbeitet man also ebenfalls mit einer abstrakten Klasse *Node*, die das geforderte Verhalten aller Knotenarten festlegt.

Ein wichtiger Vorteil von Generizität in Eiffel ist, daß der Compiler im Beispiel von *Stack* erzwingt, daß alle von *Stack* verwalteten Objekte *vom selben Typ* (homogen) sind. Wenn man mit Vererbung arbeitet, kann *Stack* auch eine heterogene Menge von Elementen verwalten, z.B. Zahlen gemischt mit Zeichenketten. Das kann erwünscht sein (siehe Kapitel 8.3) oder auch nicht. Die Homogenität der Elementmenge kann hier jedenfalls nur durch Laufzeitprüfungen sichergestellt werden.

Im *Stack*-Beispiel mit Generizität konnten wir schreiben

```
i := intStack.Pop
```

Wenn Stack mit INTEGER parametrisiert wird, liefert *Pop* immer INTEGER-Objekte. Typprüfungen können hier zur Compilezeit stattfinden. Wenn wir den Stack mit einer abstrakten Elementklasse *Node* implementieren, liefert *Pop* *Node*-Objekte, die man erst mit einer Typzusicherung in *IntegerNode*-Objekte umwandeln muß. Die Typzusicherung erfordert eine Typprüfung zur Laufzeit.

Interessant ist die Überlegung, daß man mit Vererbung Generizität simulieren kann, umgekehrt jedoch nicht. Generizität ist kein Ersatz für Vererbung [Mey86]. Vererbung ist das mächtigere und allgemeinere Konzept.

Klassen haben meist mehrere Schnittstellen

Am Beispiel der Klasse *Tree* läßt sich noch eine andere wichtige Tatsache studieren: Viele Klassen haben nicht nur eine Schnittstelle zu ihren *Klienten*, sondern meist auch eine oder mehrere Schnittstellen zu ihren *Komponenten*, in diesem Fall zu *Node* (Abb. 8.3). Der Benutzer der Klasse *Tree* muß auch die Schnittstelle ihrer Komponentenklasse *Node* kennen, denn er muß eine Erweiterung von *Node* implementieren.

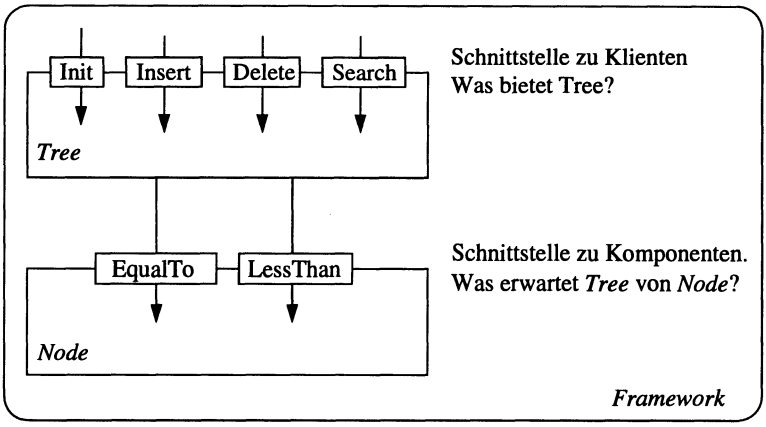


Abb. 8.3 Schnittstelle von Klassen zu Klienten und Komponenten

Wenn man eine Klasse wie *Tree* entwirft, darf man nicht nur an die Operationen denken, die die Klasse selbst anbietet. Ebenso wichtig ist es, sich zu überlegen, welche Objekte die Klasse braucht, um ihre Aufgabe zu erfüllen, und welche Operationen sie von diesen Objekten erwartet. Auf diese Weise erhält man eine Menge von Klassen, die zusammenarbeiten, um eine bestimmte Aufgabe zu erfüllen. Man nennt so ein System von Klassen ein *Framework* (siehe Kapitel 11). *Tree* und *Node* bilden ein Framework für eine Binärbaum-Verwaltung. Ein Framework stellt ein Halbfabrikat dar, das man später ausbauen kann, um es für verschiedene Zwecke zu verwenden.

Zusammenfassung

Generizität ist eine Technik, die immer dann angewendet werden kann, wenn man es mit Bausteinen zu tun hat, die andere Objekte verwalten. Will man diese Bausteine so allgemein halten, daß sie mit *verschiedenartigen* Objekten arbeiten können, dann ist eine Implementierung mittels Klassen angebracht. Man geht dabei folgendermaßen vor:

1. Überlege, welche Dienste von den verwalteten Objekten erwartet werden.
2. Entwirf eine oder mehrere abstrakte Klassen, die diese Dienste anbieten.
3. Implementiere den generischen Baustein unter Verwendung der abstrakten Klassen.

8.3 Heterogene Datenstrukturen

Eine der nützlichsten Anwendungen der objektorientierten Programmierung ist die Verwaltung *heterogener Datenstrukturen*. Situationen dieser Art sind durch folgende Merkmale gekennzeichnet:

1. Objekte treten in Varianten auf.
2. Das Programm, das die Objekte benutzt, will nicht zwischen den Varianten unterscheiden.
3. Die Anzahl der Varianten ist unbekannt. Es können später neue Varianten hinzukommen.

Tabelle 8.1 beschreibt einige Situationen dieser Art.

Konventionelle Implementierung eines Grafikeditors

Betrachten wir zum Beispiel einen Grafikeditor, mit dem man Linien, Rechtecke und Kreise zeichnen, selektieren und verschieben kann. In konventionellen Sprachen wie Modula-2 oder C würde man die Figuren durch Varianten-Records (in C Unions) implementieren:

```
TYPE
   Figure = POINTER TO FigureDesc;
   FigureDesc = RECORD
      next: Figure;
      CASE kind: FigureKind OF
         line: x0, y0, x1, y1: INTEGER
      |  rect: x, y, w, h: INTEGER
      |  circle: mx, my, radius: INTEGER
      END
   END;
```

Tabelle 8.1 Beispiele für heterogene Datenstrukturen

Varianten	Operationen
Objekte in einem Grafikeditor (Linien, Rechtecke, Kreise, ...)	zeichnen, verschieben, anklicken, ...
Objekte auf einem Bildschirm (Fenster, Ikonen, Menüs, ...)	zeichnen, verschieben, anklicken, ...
Objekte in einem Dialogfenster (Buttons, Texte, Rollbalken, ...)	zeichnen, verschieben, anklicken, ...
Objekte in einem Spiel (Jäger, Gejagte, Hindernisse, ...)	zeichnen, bewegen, kollidieren, ...
Objekte in einer Simulation (Autos, Personen, Ampeln, ...)	aktivieren, verzögern, ...

Mit Knoten dieses Typs kann man eine Liste bilden, die verschiedene Figurenarten enthält (Abb. 8.4).

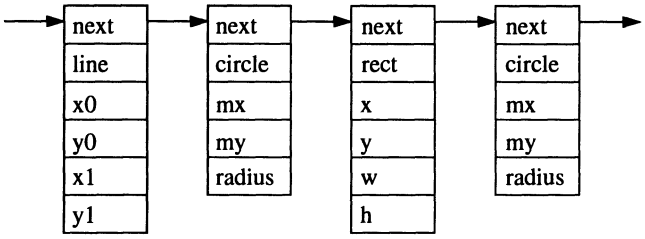


Abb. 8.4 Heterogene Datenstrukturen aus Varianten-Records

Varianten-Records sind jedoch gefährlich, weil die meisten Compiler keinen Code erzeugen, der prüft, ob Programme zur Laufzeit auf die richtige Variante eines Objekts zugreifen. Außerdem muß man bei jeder Operation auf eine Figur zwischen den einzelnen Varianten unterscheiden. Um zum Beispiel alle Figuren der Liste zu zeichnen, muß man schreiben:

```
figure := firstFigure;
WHILE figure # NIL DO (*draw all figures*)
   CASE figure^.kind OF
      line: ...    (*draw line*)
   |  rect: ...    (*draw rectangle*)
   |  circle: ...  (*draw circle*)
   END;
   figure := figure^.next
END
```

Für andere Operationen sind ähnliche Fallunterscheidungen nötig. Meist sind sie über das ganze Programm verstreut. Schlimmer noch: wenn eine neue Objektart (z.B. Spline-Kurven) hinzukommt, muß der Datentyp *Figure* geändert werden (wodurch eventuell Klientenmodule neu übersetzt werden müssen). Ferner muß in allen Fallunterscheidungen berücksichtigt werden, daß es jetzt auch Spline-Objekte gibt. Das ist mühsam und fehleranfällig. Implementierungen dieser Art sind daher unübersichtlich und schwer erweiterbar.

Objektorientierte Implementierung eines Grafikeditors

Objektorientierte Sprachen erlauben eine wesentlich elegantere Implementierung: Man betrachtet Figuren als abstrakte Objekte und fragt sich, welche Annahmen der Editor über sie machen muß, um mit ihnen arbeiten zu können. Figuren müssen verkettet werden, und man muß sie zeichnen, verschieben, lesen und schreiben können. Mehr muß der Editor nicht wissen. Die konkreten Figurenarten braucht er

gar nicht zu kennen. Diese Überlegungen führen zum Entwurf einer abstrakten Klasse *Figure* (Abb. 8.5). Die konkreten Figuren werden aus *Figure* abgeleitet. Sie enthalten zusätzliche Datenfelder und überschreiben die abstrakten Methoden von Figure.

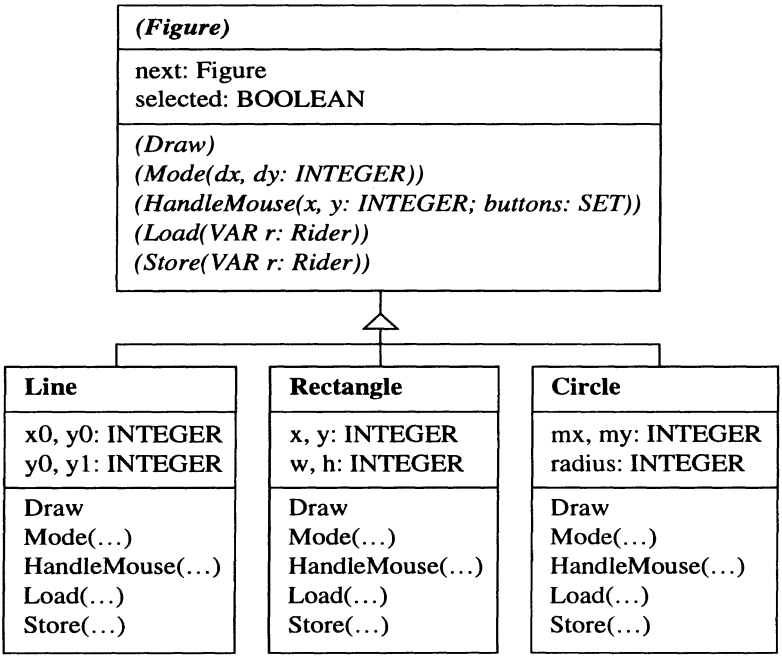


Abb. 8.5 Abstrakte Klasse *Figure* mit ihren Unterklassen

Mit Objekten dieser Art läßt sich nun ebenfalls eine heterogene Liste aufbauen (siehe Abb. 8.6). Der Editor sieht nur den gemeinsamen Teil der Objekte (den grauen Teil in Abb. 8.6). Für ihn sind alle Figuren vom statischen Typ *Figure*. In Wirklichkeit stecken hinter ihnen Objekte eines erweiterten Typs, nämlich Linien, Rechtecke und Kreise. Um alle Figuren zu zeichnen, muß der Editor nur folgendes tun:

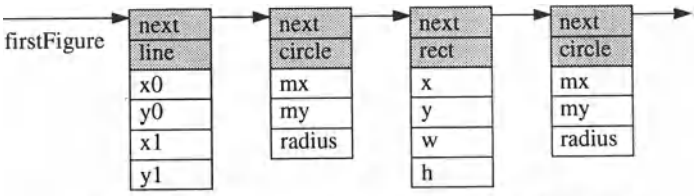


Abb. 8.6 Heterogene Datenstruktur aus Objekten

```
figure := firstFigure;
WHILE figure # NIL DO
    figure.Draw;
    figure := figure.next
END
```

Keine Fallunterscheidungen mehr nötig

Der Editor braucht sich nicht um die Varianten zu kümmern. Er schickt den Figuren einfach eine *Draw*-Meldung im Vertrauen darauf, daß jede Figur diese Meldung richtig interpretiert. Wenn später eine neue Figurenart *Spline* hinzukommt, berührt das den Editor nicht. Er kann *Spline*-Objekte in seine Datenstruktur einhängen. Wenn das Objekt, dem er eine *Draw*-Meldung schickt, ein *Spline*-Objekt ist, dann wird eben eine Spline-Kurve gezeichnet, ohne daß der Editor dafür geändert werden muß.

Bessere Lokalität

Die Operationen auf Objekte sind nun nicht mehr über das ganze Programm verstreut, sondern in den Figurenklassen gesammelt. Das vereinfacht die Wartung. Bei Einführung einer Spline-Klasse muß lediglich diese eine Klasse implementiert werden; der Rest des Programms bleibt unverändert.

Erweiterbarkeit

Man beachte, daß in diesem Beispiel zwei verschiedene Arten von Erweiterung auftreten: Erstens wurde die Klasse *Figure* zu *Line*, *Rectangle* oder *Circle* ausgebaut; zweitens wurde aber auch der ganze Editor erweitert. Ursprünglich konnte er nur mit abstrakten Figuren arbeiten, jetzt kann er auch Linien, Kreise und Rechtecke zeichnen und kann jederzeit um neue Figurenarten erweitert werden.

Ein-/Ausgabe von Figuren

Ein nichttriviales Problem ist die Ein/Ausgabe heterogener Datenstrukturen. Der Grafikeditor kann Figuren nicht selbst abspeichern und einlesen, weil er ihren inneren Aufbau nicht kennt. Er muß diese Aufgabe den einzelnen Figuren überlassen, die dazu die Methoden *Store* und *Load* überschreiben. Aber wie kann der Editor Figuren lesen, deren Struktur er gar nicht kennt? Bevor er einer Figur eine *Load*-Meldung schicken kann, muß er sie erst erzeugen. Woher weiß er, von welchem Typ sie sein muß? Dieses Problem wird in Kapitel 9.4.6 behandelt.

Laden von Erweiterungen zur Laufzeit

Zu den Besonderheiten des Oberon-Systems gehört, daß es möglich ist, Erweiterungen eines Programms erst zur Laufzeit hinzuzuladen. Implementiert man jede Figur in einem eigenen Modul, dann kann der Editor so gestartet werden, daß er zu Beginn keine einzige Unterklasse von *Figure* kennt. Auf Abb. 8.7 bezogen heißt das, daß zu Beginn nur die beiden Module *Editor* und *Figures* geladen werden. Das ergibt ein kompaktes Programm und kurze Ladezeiten.

Während der Editor läuft, kann der Benutzer nun *Lines* oder *Circles* hinzuladen und versetzt den Editor damit in die Lage, Linien oder Kreise zu zeichnen. Jeder Benutzer lädt nur so viele Teile des Pro-

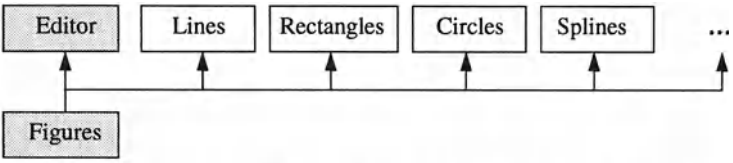


Abb. 8.7 Erweiterbarkeit von Programmen durch dynamisches Nachladen von Modulen. Rechtecke bedeuten Module, Pfeile Import-Beziehungen

gramms, wie er für seine Arbeit braucht. Es ist nicht nötig, immer alle Module zusammenzubinden und im Speicher zu haben.

Eigentlich kann man nur dann von wirklicher Erweiterbarkeit sprechen, wenn jedermann (nicht nur der Autor) ein Programm jederzeit (auch zur Laufzeit) erweitern kann. In Oberon ist das möglich. Es können neue Module (und damit neue Klassen) programmiert und dazugeladen werden, von deren Existenz das Programm nichts weiß, die es aber trotzdem benutzen kann. Der Grafikeditor weiß nichts von einem *Splines*-Modul, trotzdem kann man so ein Modul später hinzufügen, und zwar ohne daß der Editor geändert oder auch nur entladen und neu gebunden werden muß.

Zusammenfassung

Fassen wir zusammen: Wenn ein Programm mit verschiedenen Varianten von Objekten arbeiten muß, sollte es nicht zwischen ihnen unterscheiden, sondern sie als Erweiterungen ein und derselben abstrakten Klasse betrachten. Das Vorgehen ist ähnlich wie bei generischen Bausteinen:

1. Überlege, welche Operationen und Daten allen Varianten gemeinsam sind.
2. Definiere eine abstrakte Klasse mit diesen Eigenschaften.
3. Definiere für jede Variante eine konkrete Unterklasse.
4. Arbeite mit Variablen der abstrakten Klasse, ohne Rücksicht darauf, welche Varianten zur Laufzeit darin gespeichert sind.

8.4 Austauschbares Verhalten

Wenn ein Objekt oder ein Algorithmus zur Laufzeit sein Verhalten ändern soll, läßt sich das mit Klassen elegant implementieren.

Vereinheitlichung von Bildschirm- und Druckerausgabe

Sehen wir uns ein Beispiel an: Editoren stellen ihre Daten üblicherweise in einem *Rahmen* (*Frame*) am Bildschirm dar. Ein Rahmen ist eine rechteckige Zeichenfläche, auf der man Text und Grafik mit gewissen Zeichenoperationen ausgeben kann. Wenn man nun drucken

möchte, sind die gleichen Zeichenoperationen nötig, aber diesmal sollen Text und Grafik nicht auf dem Bildschirm, sondern auf dem Drucker erscheinen.

Um nicht bei jeder Zeichenoperation zwischen Bildschirm und Drucker unterscheiden zu müssen, sollte die Ausgabe nicht direkt auf den Bildschirm oder den Drucker erfolgen, sondern auf ein abstraktes Ausgabemedium, das wir *Port* nennen und das zur Laufzeit durch verschiedene konkrete Ausgabemedien ersetzt werden kann. Alle Zeichenoperationen können auf das Zeichnen von Punkten zurückgeführt werden, daher muß *Port* nur das Zeichnen von Punkten unterstützen (dies ist das austauschbare Verhalten). *Port* ist eine abstrakte Klasse und sieht wie in Abb. 8.8 gezeigt aus.

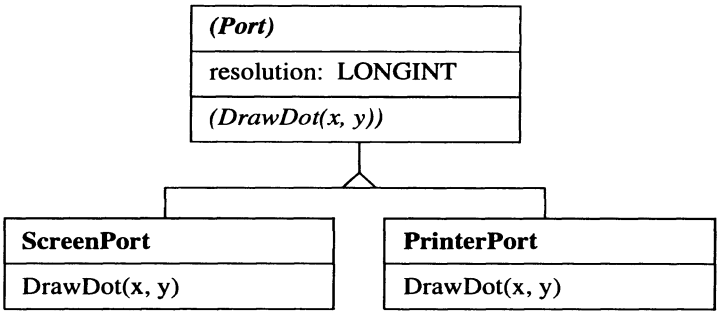


Abb. 8.8 Abstrakter Port mit seinen Unterklassen

Man nun kann verschiedene konkrete Ports definieren: einen Bildschirm-Port, der Punkte auf dem Bildschirm darstellt, oder einen Drucker-Port, der Punkte auf dem Drucker darstellt. Beides sind Unterklassen von *Port* und überschreiben dessen abstrakte Methode *DrawDot*.

Bildschirmrahmen wickeln alle Ausgaben über einen Port ab, der ein Attribut der Klasse *Frame* ist (Abb. 8.9). Je nachdem, welcher konkrete Port in *f.port* installiert wird, gehen die Ausgaben entweder auf den Bildschirm oder auf den Drucker. Man kann den Port zur Lauf-

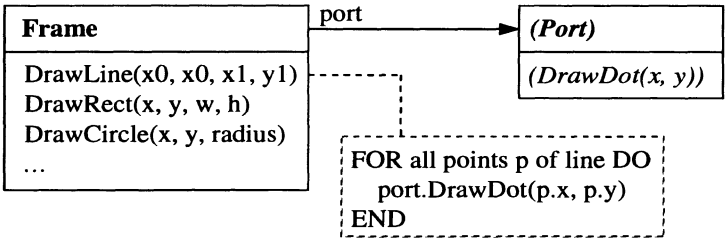


Abb. 8.9 Zeichenoperationen von *Frame* sind Medium-unabhängig

zeit wechseln und damit das Verhalten des Rahmens *f* ändern. Alle Klienten von *f*, die auf den Bildschirm ausgeben, können nun automatisch auch drucken.

Weitere Beispiele

Ein weiteres Beispiel für austauschbares Verhalten ist eine parametrisierbare Prozeßverwaltung (scheduler). Parallele Prozesse können zum Beispiel in zeitlicher Reihenfolge (first come first served; FCFS) oder nach Prioritäten bearbeitet werden. Um die Strategie jederzeit ändern zu können, ist es nützlich, die Prozeßverwaltung als Variable einer abstrakten Klasse *Scheduler* zu implementieren, die zur Laufzeit Objekte einer konkreten Klasse *FCFSScheduler* oder *PriorityScheduler* enthalten kann.

Fassen wir zusammen: Die Vorgangsweise bei austauschbarem Verhalten ist wie folgt:

1. Überlege, aus welchen Operationen das auszutauschende Verhalten besteht.
2. Definiere eine abstrakte Klasse, die diese Operationen als Methoden anbietet.
3. Implementiere konkretes Verhalten als Unterklassen davon.
4. Arbeite mit Variablen der abstrakten Klasse, die zur Laufzeit Objekte konkreter Klassen mit verschiedenem Verhalten aufnehmen können.

8.5 Anpassung bestehender Bausteine

Wir haben bereits in Kapitel 5 gesehen, daß die Wiederverwendung von Software-Bausteinen oft daran scheitert, daß die vorhandenen Bausteine nicht genau passen. Mit Klassen als Bausteinen stellt sich dieses Problem nicht. Klassen lassen sich nachträglich erweitern und anpassen. Fehlende Funktionalität kann in Unterklassen hinzugefügt werden, ohne die Basisklassen zu ändern. Anpassung von Bausteinen ist daher ebenfalls eine häufige Anwendung der objektorientierte Programmierung.

Erweiterung von Texten

Betrachten wir folgendes Beispiel: Angenommen man sucht eine Klasse, mit der man Texte samt verschiedenen Schriftarten verwalten kann. In einer Klassenbibliothek findet man eine Klasse *Text*, die zwar ASCII-Texte verwalten kann, aber keine Schriftarten (Abb. 8.10).

Die vorhandene Klasse paßt zwar nicht genau, aber es ist sicher einfacher, den gewünschten Baustein auf *Text* aufzubauen als ihn völlig neu zu schreiben. Man erspart sich dadurch Implementierungs- und Testaufwand.

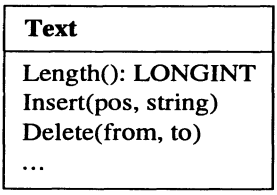


Abb. 8.10 Ein Textbaustein aus einer Bibliothek

Um verschiedene Schriftarten verwalten zu können, leitet man daher aus *Text* eine neue Klasse *StyledText* ab, die als Attribut eine Liste von Schriftarten (*Styles*) besitzt (Abb. 8.11).

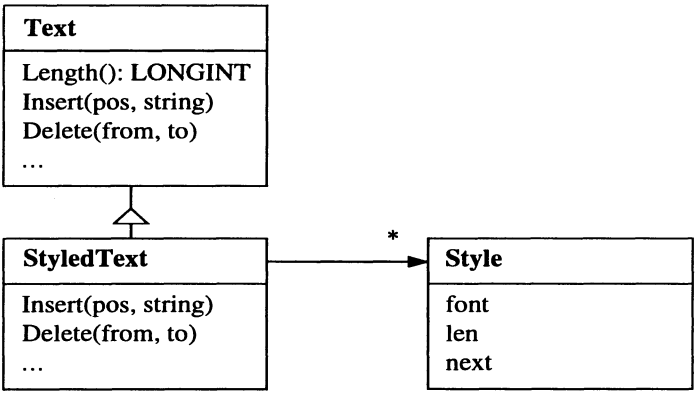


Abb. 8.11 Erweiterter Textbaustein

Der Text wird in eine Folge von Textstücken zerlegt. Für jedes Textstück gibt es einen Knoten der Klasse *Style*, der die Länge und Schriftart des Textstück angibt (Abb. 8.12).

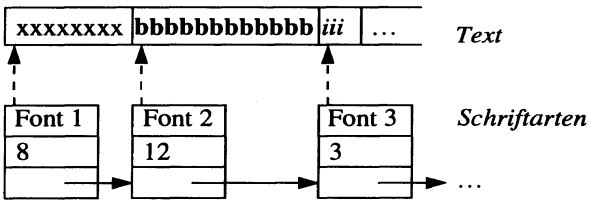


Abb. 8.12 Zuordnung von Schriftarten zu einem Text

Die Methoden *Insert* und Delete *müssen* jetzt auch auf die Schriftarten wirken, deshalb werden sie überschrieben:

```
PROCEDURE (t: StyledText) Insert (pos: LONGINT; s: ARRAY OF CHAR);
BEGIN
    ... (*update style list*)
    t.Insert^ (pos, s)  (*call Insert method from the base class*)
END Insert;

PROCEDURE (t: StyledText) Delete (from, to: LONGINT);
BEGIN
    ... (*update style list*)
    t.Delete^ (from, to)  (*call Delete method from the base class*)
END Delete;
```

Die Methode *Length* ist von den Schriftarten unabhängig und kann daher unverändert geerbt werden. Schließlich braucht man noch eine neue Methode *SetStyle*, mit der man die Schriftart eines Textstücks verändern kann.

Durch unsere Erweiterung haben wir um den vorhandenen Baustein *Text* eine neue Schicht *StyledText* gelegt und ihn so an unsere speziellen Bedürfnisse angepaßt (Abb. 8.13).

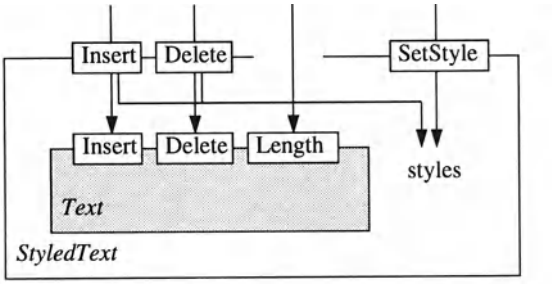


Abb. 8.13 Ummantelung von *Text* durch *StyledText*

Nach außen sieht der Baustein noch immer wie ein *Text* aus. Alle bisherigen Klienten von *Text* können daher auch mit *StyledText* arbeiten. Ein Editor, der für *Text* ausgelegt ist, merkt gar nicht, wenn man ihm *StyledText* "unterschiebt". Er kann jetzt einfach auch Schriftarten mitführen, sobald er *Insert* oder *Delete* aufruft.

Erweiterung konkreter Klassen

Während bisher immer *abstrakte* Klassen erweitert wurden, handelt es sich bei StyledText um die Erweiterung einer *konkreten* Klasse. Die Erweiterung konkreter Klassen ist eher ungewöhnlich und tritt meist dann auf, wenn sie nicht von vornherein geplant war. Eine nicht geplante Erweiterung führt aber oft zu unsauberen Lösungen, vor allem dann, wenn sie dazu mißbraucht wird, vergessene Eigenschaften einer Klasse nachträglich hinzuzufügen.

Erweiterbarkeit sollte von Anfang an geplant werden. Das heißt nicht, daß man von vornherein schon alle zukünftige Erweiterungen kennen muß. Es bedeutet lediglich, daß man sich überlegen sollte, an

welchen Stellen ein System erweiterbar sein muß, und daß man an diesen Stellen mit Variablen abstrakter Klassen arbeitet, die später Objekte beliebiger Erweiterungen enthalten können. In Kapitel 11 (Frameworks) gehen wir näher auf diese Art von geplanter Erweiterbarkeit ein.

8.6 Halbfabrikate

Ein Baustein kann bewußt unvollständig gehalten werden. Durch Verzicht auf anwendungsabhängige Teile erhöht man die Chance, ihn in anderen Programmen wiederverwenden zu können. Der Baustein enthält dann nur jene Teile, die allen Anwendungen gemeinsam sind. Er ist ein *Halbfabrikat*, das zu diversen *Endfabrikaten* ausgebaut werden kann.

Rahmen als Halbfabrikate

Ein Beispiel eines Halbfabrikats sind *Bildschirmrahmen* (Frames). In Oberon ist ein Rahmen eine rechteckige Zeichenfläche zur Darstellung von Text, Grafik oder anderen Daten. Sie kann außerdem Benutzereingaben wie Mausklicks oder Tastendrucke interpretieren und verarbeiten.

Nun gibt es verschiedene konkrete Bildschirmrahmen, die nur für bestimmte Zwecke verwendbar sind: Ein Textrahmen kann zum Beispiel nur zur Darstellung von Text verwendet werden, ein Grafikrahmen nur zur Darstellung von Grafik. Die allgemein wiederverwendbaren Teile sind lediglich jene, die allen Rahmen gemeinsam sind, nämlich:

1. Man kann einen Rahmen in einem Fenster installieren, ihn auf dem Bildschirm verschieben, ihn vergrößern und verkleinern.
2. Man kann einen Rahmen auffordern, seinen gesamten Inhalt neu zu zeichnen, wobei offen bleibt, was dieser Inhalt ist.
3. Man kann einem Rahmen Mausklicks oder eingetippte Zeichen zur Verarbeitung übergeben, wobei offen bleibt, wie der Rahmen auf die Benutzereingaben reagiert.

Ein allgemeiner Rahmen dieser Art kann zu einem Textrahmen, einem Grafikrahmen oder einem Tabellenrahmen ausgebaut werden. Er ist ein Halbfabrikat, das die anwendungsunabhängigen Teile verschiedener Rahmen-Arten zusammenfaßt und die Schnittstelle aller zukünftigen Rahmen-Erweiterungen vorgibt.

Abb. 8.14 zeigt die Schnittstelle einer solchen allgemeinen – und daher abstrakten – Klasse *Frame*. *Redraw* ist eine abstrakte Methode: Da ein abstrakter Rahmen noch nicht weiß, was er darstellen soll, bleibt sie leer. *HandleMouse* und *HandleKey* sind ebenfalls abstrakt,

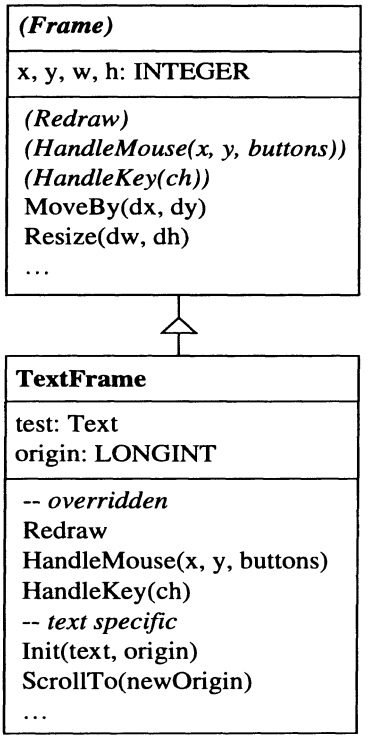


Abb. 8.14 Halbfabrikat *Frame* und Fertigprodukt *TextFrame*

denn ein Rahmen weiß noch nicht, wie er Benutzereingaben behandeln soll. Die Methode *MoveBy* ist eine konkrete Methode. Ihre Implementierung ist für alle Rahmen-Arten gleich und kann daher bereits in der Klasse *Frame* angegeben werden. *Resize* ist schließlich eine "halb konkrete" Methode: Sie verschiebt die rechte untere Ecke eines Rahmens und ruft nötigenfalls *Redraw* auf, um einen bisher unsichtbaren Teil des Rahmen-Inhalts neu zu zeichnen. Die Methode *Resize* muß in Unterklassen nicht mehr überschrieben werden. Sie leistet aber erst dann "vernünftige" Arbeit, wenn *Redraw* überschrieben wird.

Das Halbfabrikat *Frame* kann nun zu einem Fertigprodukt *TextFrame* ausgebaut werden, indem man eine Unterklasse *TextFrame* bildet, die für Texte nötigen Daten und Methoden hinzufügt und die abstrakten Methoden von *Frame* überschreibt. Dies wird ebenfalls in Abb. 8.14 gezeigt.

Textrahmen erben das gesamte Verhalten von Rahmen: man kann sie in einem Fenster installieren, das Fenster fordert sie auf, sich bei

Bedarf neu zu zeichnen und das System schickt ihnen Meldungen wenn eine Benutzereingabe erfolgt.

Auf die gleiche Weise kann man eine Klasse *GraphicFrame* oder *TableFrame* ableiten. Man muß dabei nicht bei Null beginnen, sondern kann den Entwurf und die Implementierung des Halbfabrikats *Frame* wiederverwenden. Dabei ist eine Dokumentation wichtig, die einem sagt, welche Methoden man überschreiben muß und welche nicht.

8.7 Zusammenfassung

Dieses Kapitel zeigte einige typische Situationen, in denen sich der Einsatz von Klassen lohnt. Der Leser möge sich diese Situationen und ihre Lösungen einprägen.

Zusammenfassend läßt sich sagen: Objektorientierte Programmierung ist immer dann angebracht, wenn man es mit *komplexen Objekten* zu tun hat, insbesondere wenn diese *in Varianten auftreten*, zwischen denen man bei gemeinsamen Operationen nicht unterscheiden will.

Objektorientierte Programmierung ist ferner für Systeme mit *hohen Anforderungen an Erweiterbarkeit* geeignet. Bei einem Grafikeditor muß es zum Beispiel möglich sein, zu einem späteren Zeitpunkt eine neue Figurenart hinzuzufügen, deren Exemplare wie alle anderen Figuren dargestellt und verschoben werden können, ohne daß man die bestehende Software ändern muß.

Schließlich ist objektorientierte Programmierung für die Implementierung von *Bibliotheksbausteinen* geeignet. Wenn man schon Bausteine in einer Bibliothek sammelt, kann es nur von Vorteil sein, wenn man sie in Form von Klassen erweiterbar und anpaßbar macht.

Ziel der objektorientierten Programmierung ist nicht, passende Bausteine für eine *bestimmte* Anwendung herzustellen, sondern Bausteine, die *möglichst oft wiederverwendet* werden können. Insbesondere ist es wichtig, gute Abstraktionen zu finden, aus denen sich möglichst viele konkrete Klassen ableiten lassen.

9 Entwurfsmuster

Erfahrene Programmierer zeichnen sich dadurch aus, daß sie über ein gewisses *Lösungsrepertoire* für häufig auftretende Probleme verfügen. Dies macht ihren *Erfahrungsschatz* aus. Wenn sie vor einer bestimmten Aufgabe stehen, müssen sie sich die Lösung nicht selbst erarbeiten, sondern können auf eine bewährte Lösung aus der Vergangenheit zurückgreifen.

Standardlösungen dieser Art nennt man *Entwurfsmuster* (*design patterns*). Sie stellen schematische Lösungen für häufig wiederkehrende Probleme dar. In der objektorientierten Programmierung wurden Entwurfsmuster durch das ausgezeichnete Buch von Gamma et al. [GHJV96] bekannt gemacht (siehe auch [Pre95] und [BMRSS96]). Sie sind aber nicht auf die objektorientierte Programmierung beschränkt und wurden auch nicht von ihr erfunden. Entwurfsmuster sind nichts anderes als die Algorithmen und Datenstrukturen der objektorientierten Programmierung. Der Begriff Objektstrukturen wäre daher für sie nicht unpassend.

9.1 Motivation

Muster in konventionellen Programmen

Bevor wir in diesem Kapitel eine Sammlung nützlicher Entwurfsmuster besprechen, wollen wir kurz ihr Wesen und ihre Vorteile studieren. Dabei wählen wir als Beispiel ein Muster aus der konventionellen Programmierung. Jeder Programmierer kennt wohl die in Abb. 9.1 dargestellte Datenstruktur. Es handelt sich um einen Binärbaum und

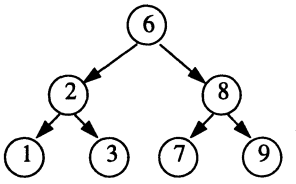


Abb. 9.1 Ein Binärbaum als Beispiel eines Entwurfsmusters

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

stellt ein typisches *Muster* dar. Binärbäume dienen bekanntlich zum effizienten Suchen von Elementen in einer großen Datenmenge. Wann immer man vor der Aufgabe steht, effizient suchen zu müssen, kann man auf das bewährte Muster Binärbaum zurückgreifen.

Bestandteile eines Musters

Welche Bestandteile machen ein Muster aus? Im wesentlichen besteht es aus einem *Namen*, unter dem man es kennt, aus der *Beschreibung des Problems*, für das man es anwenden kann, und aus der *eigentlichen Lösung* des Problems.

Der *Name* schafft ein griffiges und vertrautes Vokabular, das Entwickler verwenden können, wenn sie miteinander kommunizieren. Es ist einfacher zu sagen "Nimm einen Binärbaum", als "Organisiere die Elemente der Datenmenge so, daß jedes Element bis zu zwei Söhne hat und sein Wert größer als der Wert seines linken Sohnes und kleiner oder gleich als der Wert seines rechten Sohnes ist".

Die *Problembeschreibung* lautet für den Binärbaum: Man verwendet einen Binärbaum, wenn man eine große Datenmenge im Hauptspeicher verwalten muß und darin effizient einfügen, löschen und suchen will. Die Problembeschreibung kann auch auf Einschränkungen und Gefahren hinweisen. Binärbäume sind zum Beispiel weniger gut geeignet, wenn sich die Datenmenge häufig ändert, weil dabei die Gefahr besteht, daß der Baum degeneriert.

Die dritte Komponente eines Musters ist seine *Realisierung*. Man beschreibt sie oft bewußt abstrakt und geht nicht auf die detaillierte Implementierung ein. Dadurch wird das Muster unabhängig von einer bestimmten Programmiersprache oder einer konkreten Datenstruktur. Die Realisierung eines Binärbaums beruht z.B. ausschließlich auf der Ordnung, die zwischen jedem Knoten und seinen Söhnen definiert ist.

Muster in objektorientierten Programmen

Entwurfsmuster im *objektorientierten Sinn* behandeln das Zusammenspiel von *Klassen* oder besser gesagt Objekten bei der Lösung eines gegebenen Problems. Sie werden daher meist durch Klassendiagramme beschrieben. Weil es oft nötig ist, das dynamische Zusammenspiel der Klassen zu beschreiben, fügt man den Klassendiagrammen oft kleine Codestücke hinzu oder beschreibt ihr Verhalten durch ein Interaktionsdiagramm.

Gamma [GHJV96] gibt einen Katalog von etwa 25 Mustern an. Wir gehen in diesem Buch nur auf die wichtigsten davon ein, insbesondere auch deshalb, weil viele Muster einander ähnlich sind. Dafür stellen wir einige andere Muster vor, die nicht im Buch von Gamma vorkommen, die sich aber aus unserer Sicht bewährt haben.

Gamma teilt die Muster in drei Kategorien ein: *Erzeugende Muster* behandeln die flexible Erzeugung von Objekten oder Objektstrukturen; *Strukturmuster* behandeln häufige Verknüpfungen von Objekten zu größeren Strukturen; *Verhaltensmuster* schließlich beschreiben häufig auftretendes dynamisches Verhalten von Objekten.

9.2 Erzeugende Muster

Objekte werden meist dynamisch angelegt. In Oberon-2 gibt es dafür die Standardprozedur NEW, andere Programmiersprachen bieten ähnliche Erzeugungsoperatoren an. So einfach wie die Erzeugung eines Objekts klingt, ist sie in der Praxis aber oft nicht. Manchmal müssen zusätzliche Aktionen ausgeführt werden (z. B eine Initialisierung), oft weiß man statisch auch gar nicht, von welchem Typ das zu erzeugende Objekt sein soll. Wir besprechen daher in diesem Kapitel drei erzeugende Muster, die auf diese Situationen eingehen:

- Konstruktor — Erzeugung und Initialisierung von Objekten
- Fabrik — Erzeugung von Objekten variablen Typs
- Prototyp — Erzeugung von Objekten variablen Typs

9.2.1 Konstruktor

Konstruktor

Bei der Erzeugung eines Objekts möchte man auch gleich seine Attribute initialisieren. Um die Initialisierung nicht zu vergessen, ist es sinnvoll, die Erzeugung und Initialisierung zu einer einzigen Aktion zusammenzufassen. Dies ist der Zweck des Konstruktor-Musters.

Viele Programmiersprachen (z.B. C++ oder Java) besitzen Konstruktoren bereits als Teil der Sprache. Oberon-2 hat kein solches Sprachkonstrukt, man kann es aber leicht durch das Konstruktor-Muster nachbauen.

Betrachten wir eine Klasse *T* mit einem Attribut *a* und einer Methode *InitT*, die den Wert von *a* initialisiert (Abb. 9.2). Ein Konstruktor für *T* wird nun als Funktionsprozedur *NewT* implementiert, die ein *T*-Objekt erzeugt und seine Initialisierungsprozedur *InitT* aufruft:

```
PROCEDURE NewT(x: ...): T;
   VAR t: T;
BEGIN
   NEW(t); t.InitT(x);
   RETURN t
END NewT;
```

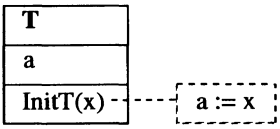


Abb. 9.2 Eine Klasse mit Initialisierungsmethode

Wann immer man nun ein *T*-Objekt braucht, erzeugt und initialisiert man es nun durch Aufruf seines Konstruktors:

```
obj := NewT(x)
```

Dieses Muster ist leicht auf beliebige Klassen übertragbar, indem man für *T* den jeweiligen Klassennamen einsetzt (z.B. *NewRectangle* und *InitRectangle*).

Konstruktor für Unterklassen

Wenn man ein Objekt einer Unterklasse erzeugt, müssen nicht nur die Attribute dieser Klasse, sondern auch die Attribute der Basisklasse(n) initialisiert werden. Das Konstruktor-Muster läßt sich leicht für diesen Zweck anpassen (Abb. 9.3). Wenn *S* eine Unterklasse von *T* ist, dann implementiert man einen Konstruktor *NewS*, dessen Aufruf

```
obj := NewS(x, y)
```

ein neues *S*-Objekt erzeugt und es durch Aufruf von *InitS* initialisiert. Dabei werden auch die Basisattribute durch Aufruf von *InitT* initialisiert.

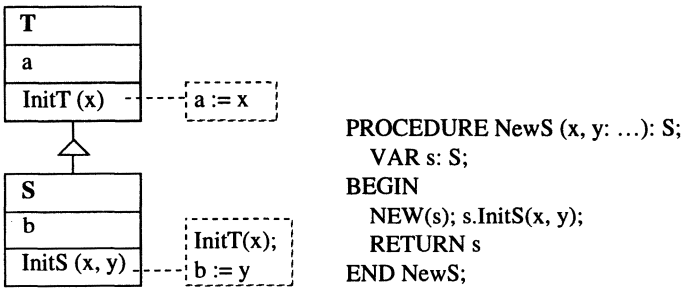


Abb. 9.3 Konstruktor einer Unterklasse

9.2.2 Fabrik

Fabrik

Das Fabrik-Muster dient zur Erzeugung von Objekten, deren dynamischen Typ man nicht statisch im Programm festlegen möchte.

Am besten sehen wir uns das an Hand eines Beispiels an. Nehmen wir an, es gibt verschiedene Arten von Texten, die aus einer abstrakten Klasse *Text* abgeleitet sind (Abb. 9.4). *SimpleText* stellt einfache ASCII-Texte dar, während *StyledText* auch Schriftarten verwaltet.

Stellen wir uns nun vor, daß ein Editor mit diesen Texten arbeiten will, wobei es möglich sein soll, wahlweise einen *SimpleText* oder einen *StyledText* zu verwenden. Der Editor hat dazu ein Attribut *t* der

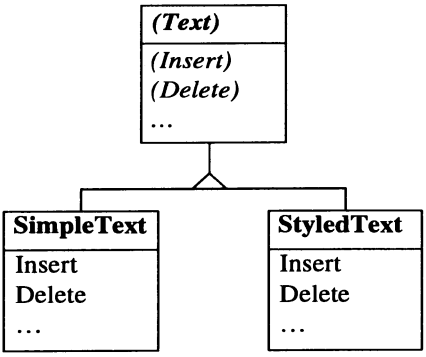


Abb. 9.4 Verschiedene Arten von Texten

abstrakten Klasse *Text*, in dem er sowohl *SimpleText*-Objekte als auch *StyledText*-Objekte speichern kann (Abb. 9.5).

Beim Öffnen eines Editorfensters in der Methode *Open* wird ein neues *Text*-Objekt erzeugt. Dabei muß sich der Editor festlegen, von welchem Typ dieser *Text* sein soll. In Abb. 9.5 erzeugt der Editor mittels eines Konstruktors ein *SimpleText*-Objekt. Diese Entscheidung ist fest in den Code eingebrannt. Will man *StyledText*-Objekte erzeugen, muß man den Code der Methode *Open* ändern.

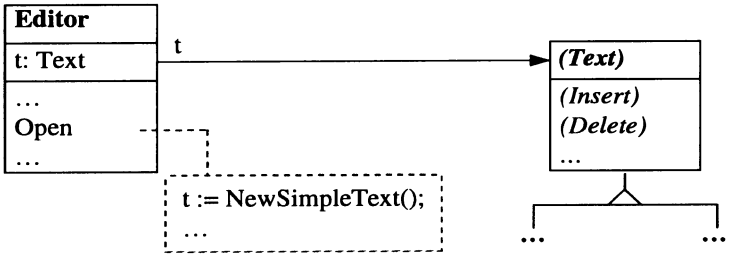


Abb. 9.5 Der dynamische Typ von *t* wird statisch festgelegt

Das ist genau das Problem, welches durch das Fabrik-Muster gelöst wird. Anstatt *statisch* festzulegen, von welchem Typ der Text sein soll, überläßt man die Erzeugung des Textes einem *Fabrik-Objekt*. Für jede Textart gibt es eine eigene Fabrik (*SimpleFactory*, *StyledFactory*, etc.), die Objekte von dieser Textart erzeugt. Alle Fabriken sind von einer abstrakten Fabrik-Klasse abgeleitet, so daß sich das in Abb. 9.6 gezeigte Muster ergibt.

Bei der Initialisierung des Editors weist man dem Attribut *factory* ein *SimpleFactory*- oder ein *StyledFactory*-Objekt zu. Wählt man zum Beispiel ein *StyledFactory*-Objekt, führt der Aufruf

```
t := factory.New()
```

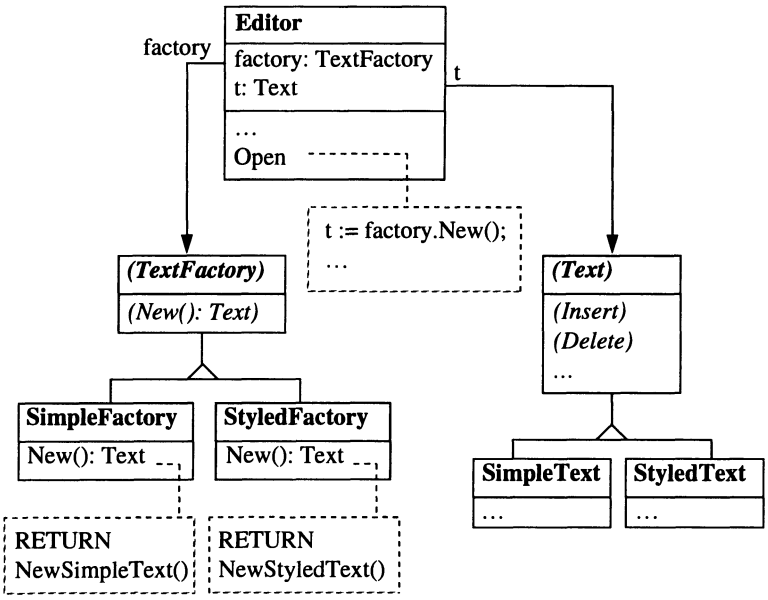


Abb. 9.6 Erzeugung eines Textes durch ein Fabrik-Objekt

in der Methode *Open* zum Aufruf der *New*-Methode von *StyledFactory*. Dort wird ein *StyledText*-Objekt erzeugt und an den Editor zurückgegeben.

Der Code von *Open* legt nun nicht mehr statisch fest, welcher Text erzeugt wird. Der Text wird von einer Fabrik geliefert, die man bei der Initialisierung des Editors auswählen und später zur Laufzeit sogar ändern kann.

Fassen wir also zusammen: wenn man Objekte erzeugen möchte, deren dynamischen Typ man flexibel halten will, erzeugt man sie nicht selbst, sondern fordert sie von einem Fabrik-Objekt an. Man kann unterschiedliche Arten von Fabriken vorsehen, die unterschiedliche Objekte liefern. Die gewünschte Art der Fabrik legt man bei der Initialisierung des Systems fest.

9.2.3 Prototyp

Prototyp

Der Zweck des Prototyp-Musters ähnelt dem des Fabrik-Musters. Man verwendet es, um Objekte zu erzeugen, deren dynamischen Typ man flexibel halten will. Die Realisierung eines Prototyps unterscheidet sich jedoch von der einer Fabrik und ist in vieler Hinsicht einfacher.

Wenn man ein neues Objekt eines bestimmten Typs braucht, erzeugt man es nicht selbst, sondern kopiert ein *Prototyp-Objekt*, das bereits den gewünschten Typ hat.

Bleiben wir bei unserem Editor-Beispiel aus Kapitel 9.2.2. In diesem Beispiel gibt es verschiedene Textarten, die man wahlweise im Editor verwenden möchte. Von jeder Textart legt man ein Prototyp-Objekt an (also ein *SimpleText*-Objekt und ein *StyledText*-Objekt). Eines dieser Prototyp-Objekte speichert man bei der Initialisierung des Editors im Attribut *protoText*. In der *Open*-Methode des Editors erzeugt man einfach eine Kopie von *protoText* und benutzt diese als Text im Editorfenster (Abb. 9.7).

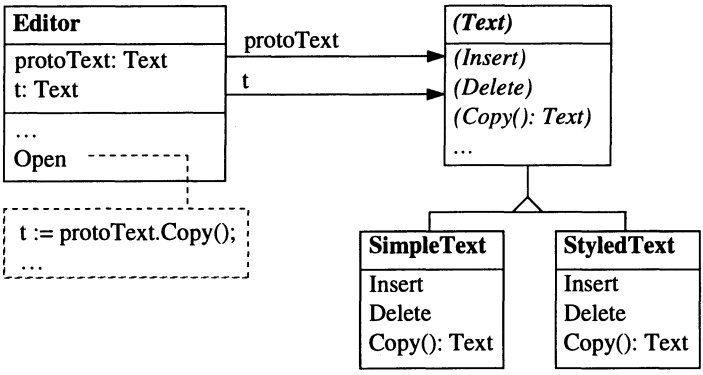


Abb. 9.7 Erzeugung eines Textes durch Kopieren eines Prototyp-Objekts

Prototyp versus Fabrik

Wie unterscheidet sich das Prototyp-Muster vom Fabrik-Muster? Das Prototyp-Muster setzt voraus, daß die zu erzeugenden Objekte (hier die *Text*-Objekte) eine Kopie von sich liefern können, also eine *Copy*-Methode besitzen. Das ist keine große Einschränkung, mag jedoch bei bereits vorhandenen Klassenhierarchien nicht immer gegeben sein. Das Fabrik-Muster hat diese Einschränkung nicht und kann bei der Erzeugung der Objekte auch gleich eine passende Initialisierung vornehmen, die zum Beispiel vom momentanen Programmzustand abhängt. Dafür hat das Fabrik-Muster den Nachteil, daß man eine eigene Klassenhierarchie für Fabriken braucht, was das Gesamtsystem komplizierter macht.

Ein weiterer Vorteil des Prototyp-Musters ist, daß man damit nicht nur eine Kopie eines *einzelnen* Objekts erzeugen kann, sondern sogar eine Kopie eines ganzen *Subsystems* aus mehreren Objekten. Man kann solche Subsysteme zur Laufzeit zusammenstellen und erhält dann bei jeder *Copy*-Operation eine Kopie des gesamten Subsystems. Dies wäre mit Fabrik-Objekten schwerer zu realisieren.

9.3 Strukturmuster

Objekte kommen selten isoliert vor, sondern arbeiten mit anderen Objekten zusammen, um irgendeine Aufgabe zu erfüllen. Dabei bilden sie gewissen Strukturen, die häufig in dieser Form auftreten und die wir unter der Kategorie der Strukturmuster beschreiben. Wir beschränken uns dabei auf folgende Muster:

- Familie — Bildung von Klassenhierarchien
- Adapter — Anpassung einer fremden Klasse an eine Familie
- Kompositum — Zusammenfassung von Teilen zu einem neuen Teil
- Dekorator — Hintereinanderschalten von Funktionalität
- Zwilling — Vermeidung mehrfacher Vererbung

9.3.1 Familie

Familie

Die Familie ist ein sehr einfaches – ja fast triviales – Muster. Wir erwähnen es nur, um für dieses Muster einen Namen zu definieren. Eine Familie besteht aus einer abstrakten Klasse und ihren Unterklassen. Abb. 9.8 zeigt eine Familie von GUI-Objekten, Abb. 9.4 eine Familie von Text-Objekten.

Alle Mitglieder einer Familie weisen dieselbe Schnittstelle auf wie ihre abstrakte Klasse. Man kann daher die einzelnen Mitglieder der Familie gegeneinander austauschen. Ein Programm, das mit GUI-Objekten arbeiten kann, kann auch mit *Checkbox*, *Button* und *Scrollbar* arbeiten.

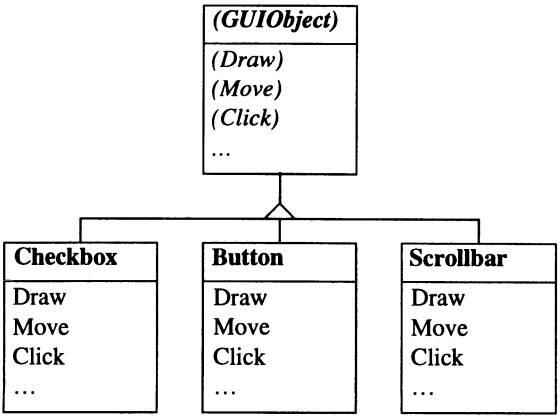


Abb. 9.8 Familie von GUI-Objekten

9.3.2 Adapter

Adapter

Manchmal möchte man eine bereits existierende Klasse zu einer bestimmten Familie kompatibel machen. Mit anderen Worten: man möchte sie wie ein Mitglied dieser Familie betrachten. Dazu müßte man sie von der abstrakten Basisklasse der Famile ableiten, was oft nicht möglich ist, weil die Klasse bereits von einer anderen Basisklasse abgeleitet ist und man keine mehrfache Vererbung verwenden kann oder will. Außerdem kann es sein, daß man den Quellcode der Klasse gar nicht besitzt und daher ihre Vererbungsbeziehung nicht ändern kann.

Die Lösung dieses Problems besteht darin, ein neues Mitglied der Familie einzuführen, das als *Adapter* zur fremden Klasse wirkt. Es implementiert die Meldungen der Familie durch Meldungen an die fremde Klasse und leitet sie somit um.

Betrachten wir ein Beispiel: Ein Grafikeditor verwaltet eine Familie von Figuren (Linien, Rechtecke, Kreise, etc.). Nun soll es auch möglich sein, *Texte* im Grafikeditor zu bearbeiten. Nehmen wir an, daß es bereits eine Klasse *Text* gibt, die jedoch kein Mitglied der Figuren-Familie ist, also nicht wie eine Figur behandelt werden kann. Daher müssen wir einen *TextAdapter* einführen, der *Figure*-Meldungen auf *Text*-Meldungen abbildet (Abb. 9.9). Die Methode *Draw* des Text-Adapters wird zum Beispiel so implementiert, daß der Text zeichenweise gelesen und auf den Bildschirm geschrieben wird.

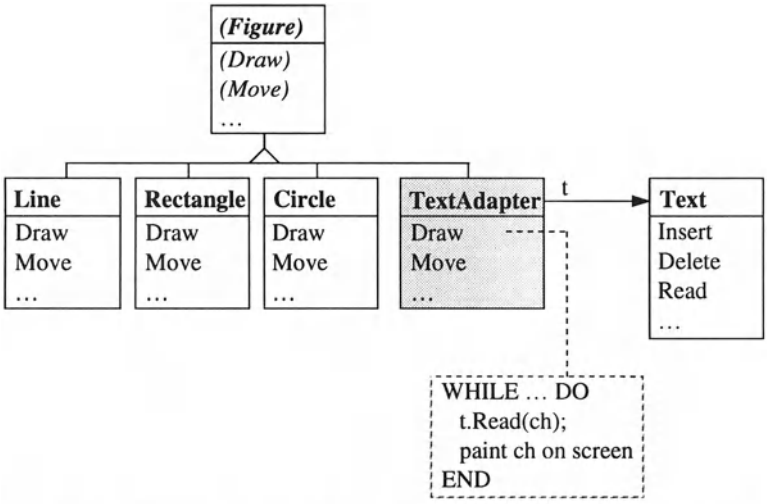


Abb. 9.9 Anpassung einer *Text*-Klasse an die *Figure*-Familie mittels eines *Text*-Adapters

Wrapper

Man kann den Text-Adapter auch als eine Schicht sehen, die die *Text*-Klasse umhüllt und ihr eine andere Schnittstelle gibt. Aus diesem Grund wird ein Adapter manchmal auch als *Wrapper* bezeichnet (to wrap = einwickeln, verpacken).

Das Adapter-Muster ist eines der am häufigsten verwendbaren Muster. In fast allen objektorientierten Programmen steht man irgendwo vor der Situation, zwei bisher nicht verwandte Klassen zueinander kompatibel zu machen. Der Adapter löst dieses Problem auf einfache Art.

9.3.3 Kompositum

Kompositum

Oft möchte man mehrere Einzelobjekte zu einem größeren Objekt zusammenfassen, das sich wieder wie ein Einzelobjekt verhält, sich also zum Beispiel wieder mit anderen Objekten zu einem noch größeren Objekt vereinigen läßt.

Die Struktur, die sich aus der rekursiven Gruppierung von Einzelobjekten ergibt, nennt man *Kompositum* (*composite*). Sie kommt zum Beispiel in einem Grafikeditor vor, in dem man mehrere Einzelfiguren zu einer Gruppe verschmelzen möchte, die dann wie ein Einzelobjekt gezeichnet, verschoben oder kopiert werden kann. Abb. 9.10 zeigt diese Situation.

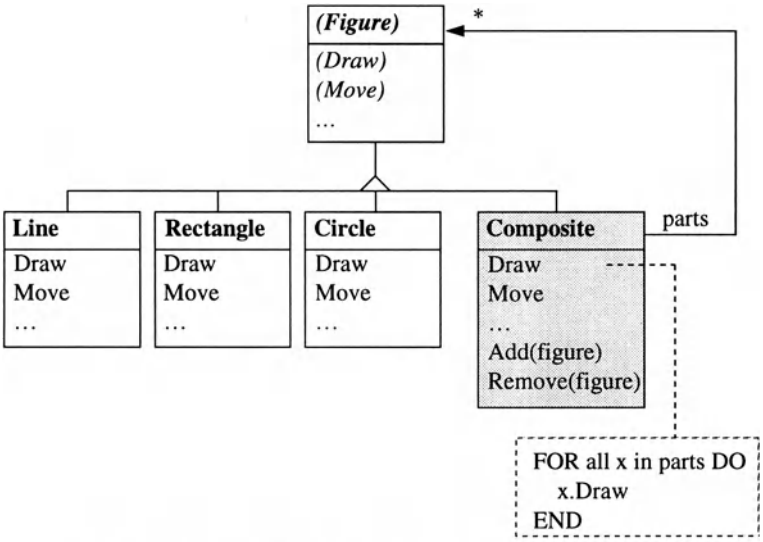


Abb. 9.10 Kompositum zum Gruppieren von Figuren in einem Grafikeditor

Composite ist eine Unterklasse von *Figure*, kann also wie eine Figur behandelt werden. Es besteht aus einer Menge von Figuren (*parts*), wobei diese wiederum *Composites* sein können. Die Meldungen an *Composite* werden auf Meldungen an die Teilobjekte abgebildet. Eine *Draw*-Meldung an die Gruppe führt also zu mehreren *Draw*-Meldungen an die Teile (*parts*). Oft ist es nützlich, auch einen Zeiger von den Teilobjekten zum Kompositum zu haben. Dieser Zeiger ist zweckmäßigerweise ein Attribut von *Figure*.

Composite besitzt noch eine *Add*- und eine *Remove*-Methode, mit denen Einzelfiguren zur Gruppe hinzugefügt oder aus der Gruppe entfernt werden können. In [GHJV95] sind *Add* und *Remove* Methoden der abstrakten Basisklasse von *Composite* (hier von *Figure*), was aber unnatürlich erscheint, weil sie für Einzelobjekte nicht sinnvoll implementiert werden können.

Auch für das Kompositum-Muster gibt es zahlreiche Anwendungen. Es kommt zum Beispiel in einer Fensterverwaltung vor, bei der einzelne Fenster Unterfenster enthalten können. Ein anderes Beispiel ist ein Formeleditor, in dem eine Formel (z.B. ein Integral) aus Unterformeln (z.B. Brüchen) bestehen kann und diese wiederum aus Unterformeln, usw.

9.3.4 Dekorator

Dekorator

Das Dekorator-Muster dient dazu, neues Verhalten zu einer gegebenen Klasse hinzuzufügen, ohne die Klasse zu ändern oder eine Unterklasse von ihr abzuleiten. Das neue Verhalten wird einfach *vor die Klasse geschaltet* und läßt sich sogar zur Laufzeit hinzufügen oder wieder entfernen.

Zur Erläuterung dieses Musters wählen wir ein Beispiel aus [GHJV95]. Ein Fenstersystem benutzt sogenannte *Frames* als rechteckige Zeichenflächen, in denen Text oder Grafik dargestellt werden kann. Es gibt also eine *Frame*-Familie mit Unterklassen wie *TextFrame* oder *GraphicFrame*. Nun möchte man zu diesen Frames verschiedene Eigenschaften hinzufügen. Zum Beispiel möchte man auf Wunsch Rollbalken (scrollbars) haben, mit denen man den Inhalt des Frames verschieben kann. Ein anderer Wunsch ist die Umrandung des Frames mit einer dicken Linie.

Würde man diese Eigenschaften durch Unterklassen realisieren, käme es wegen der großen Anzahl von Kombinationen möglicher Erweiterungen rasch zu einer unübersichtlichen Klassenhierarchie. Abb. 9.11 zeigt, was passieren würde, wenn man Frames mit Rollbalken, Frames mit Rand sowie Frames mit Rollbalken und Rand als eigene Unterklassen implementieren würde.

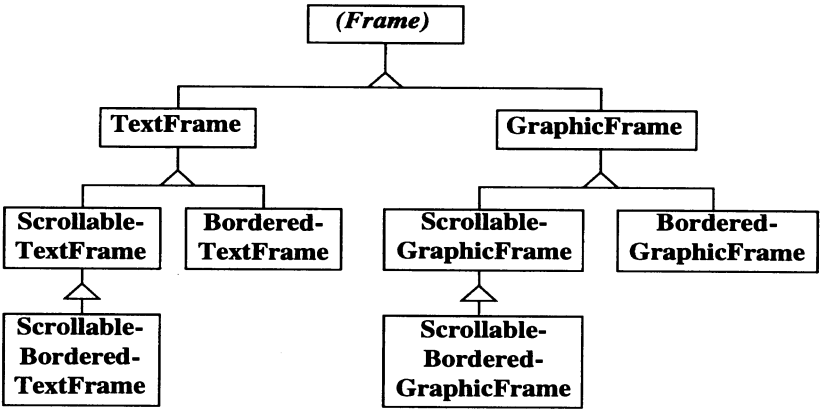


Abb. 9.11 Explosion der Klassenanzahl bei Implementierung neuer Eigenschaften als Unterklassen

Die Idee des Dekorator-Musters ist daher, neues Verhalten einer Klasse nicht durch eine Unterklasse zu realisieren, sondern durch eine *davorgeschaltete* Klasse. Abb. 9.12 zeigt, wie das für das obige Beispiel funktioniert. Einem *TextFrame*-Objekt ist ein *ScrollDecorator*-Objekt vorgeschaltet. Wenn dieses eine *Draw*-Meldung empfängt, zeichnet es einen Rollbalken und leitet dann die *Draw*-Meldung an das *TextFrame*-Objekt weiter, damit dieses seinen Text zeichnet.

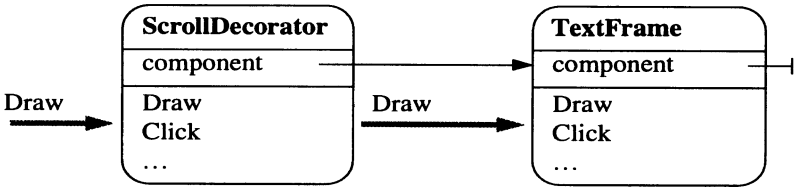


Abb. 9.12 Vor ein *TextFrame*-Objekt geschaltetes *ScrollDecorator*-Objekt

Alle Klienten, die bisher mit dem *TextFrame*-Objekt arbeiteten, müssen ihre Meldungen nun an das *ScrollDecorator*-Objekt schicken. Dieses wirkt wie ein Stellvertreter des *TextFrame*-Objekts und muß daher dieselbe Schnittstelle aufweisen, oder anders gesagt, zur selben Familie gehören wie das *TextFrame*-Objekt. Somit ergibt sich das in Abb. 9.13 gezeigte Klassendiagramm des Dekorator-Musters.

Die Klasse *Decorator* gehört zur *Frame*-Familie, kann also wie jeder andere Frame behandelt werden. Über das Attribut *component* kann ein *Decorator* mit einem anderen Mitglied der *Frame*-Familie verknüpft werden, insbesondere auch mit einem weiteren *Decorator*.

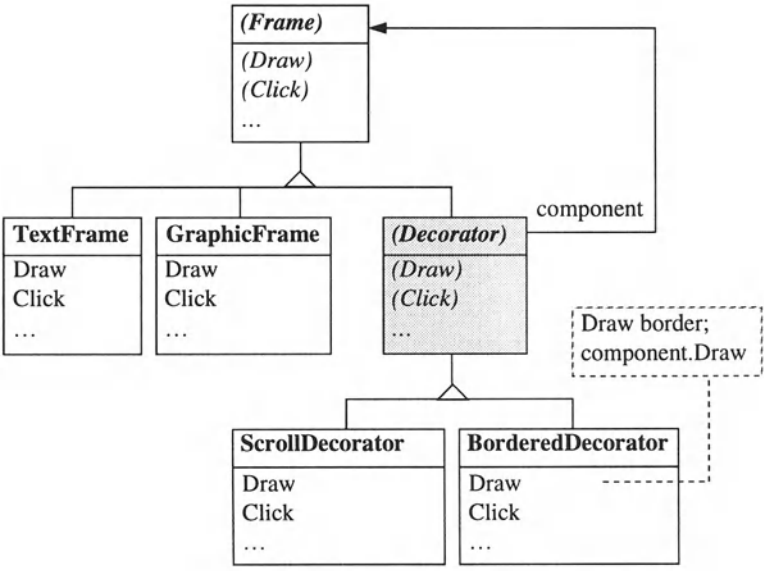


Abb. 9.13 Dekorator-Muster

So ist es zum Beispiel möglich, vor einen *GraphicFrame* sowohl einen *BorderedDecorator* als auch einen *ScrollDecorator* zu schalten, wodurch beide Eigenschaften kombiniert werden. *BorderedDecorator* zeichnet beim Erhalt einer *Draw*-Meldung eine Umrandung und leitet die *Draw*-Meldung dann an den *ScrollDecorator* weiter. Dieser zeichnet Rollbalken und leitet die *Draw*-Meldung seinerseits an das *GraphicFrame*-Objekt weiter.

Man beachte, daß nicht nur alle Eigenschaften beliebig miteinander kombiniert werden können, ohne eine Unzahl von Unterklassen zu benötigen, sondern daß die Kombination sogar zur Laufzeit änderbar ist. Zum Beispiel kann man die Umrandung des Frames erst dann hinzufügen, wenn sich der Mauszeiger in diesem Frame befindet. Vererbung ist in dieser Hinsicht weniger flexibel. Die Vererbungsbeziehung läßt sich zur Laufzeit nicht ändern.

Das Hauptproblem des Dekorator-Musters ist, daß die Identität des dekorierten Objekts verloren geht. Klienten referenzieren nicht mehr das *Frame*-Objekt, sondern das *Decorator*-Objekt. Wenn man dieses dynamisch davorschaltet, muß man aufpassen, daß alle bestehenden Zeiger von Klienten auf das *Frame*-Objekt richtiggestellt werden. Klienten können auch auf die Attribute des *Frame*-Objekts nicht mehr direkt zugreifen, weil sie dieses Objekt nicht mehr direkt referenzieren.

Orthogonale Erweiterbarkeit

Das Dekorator-Muster ist äußerst mächtig. Eine leicht abgewandelte Anwendung dieses Musters erlaubt es, eine Klasse in mehrere Rich-

tungen (orthogonal) zu erweitern. Sehen wir uns das wieder an einem Beispiel an.

In Kapitel 6 haben wir eine abstrakte Klasse *Stream* besprochen, von der es verschiedene Erweiterungen wie *Terminal*, *File* oder *Network* gab. Dies bot uns Erweiterbarkeit in *eine* bestimmte Richtung, nämlich hinsichtlich des Ausgabemediums. Wenn wir nun eine *zweite* Erweiterungsrichtung vorsehen wollen, die unterschiedliche Verschlüsselungsarten des *Stream*-Inhalts zuläßt (z.B. RSA-Verschlüsselung, DES-Verschlüsselung, etc), dann ergibt sich dadurch eine zweite (orthogonale) Dimension. Wie man aus Abb. 9.14 sieht, kann man jedes Ausgabemedium mit jeder Verschlüsselungstechnik kombinieren. Würde man diese Kombinationen mit Unterklassen realisieren, dann bräuchte man für jeden Kreuzungspunkt des Gitters eine eigene Unterklasse von Stream (also *RSATerminal*, *RSAFile*, *RSANetwork*, etc). Die Erweiterbarkeit wäre dadurch behindert. Würde man zum Beispiel ein neues Ausgabemedium einführen, dann müßte man es sofort mit allen möglichen Verschlüsselungsarten kombinieren, was zu einer großen Zahl neuer Unterklassen führen würde.

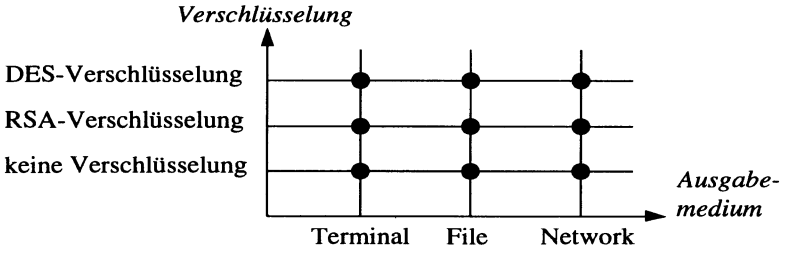


Abb. 9.14 Erweiterung einer *Stream*-Klasse hinsichtlich Ausgabemedium und Verschlüsselungstechnik

Die Explosion der Klassenanzahl kann wieder mit dem Dekorator-Muster verhindert werden. Die Verschlüsselungstechnik wird als Dekorator vor die jeweilige *Stream*-Klasse geschaltet. In Abb. 9.15 übernimmt die Klasse *Encoder* die Rolle des Dekorators.

Um eine DES-Verschlüsselung mit einem *File* zu kombinieren, hängt man ein *DESEncoder*-Objekt vor ein *File*-Objekt. Bei einer *Write*-Meldung an den *DESEncoder* wird das Zeichen *ch* verschlüsselt und anschließend mit *stream.Write(ch)* an das dahinter hängende *File*-Objekt weitergereicht. Auf diese Weise bleiben Streams in beide Richtungen erweiterbar. Man kann jederzeit neue Speichermedien (z.B. *MemoryFile*) oder neue Verschlüsselungstechniken hinzufügen und kann trotzdem jedes Ausgabemedium mit jeder Verschlüsselungstechnik kombinieren.

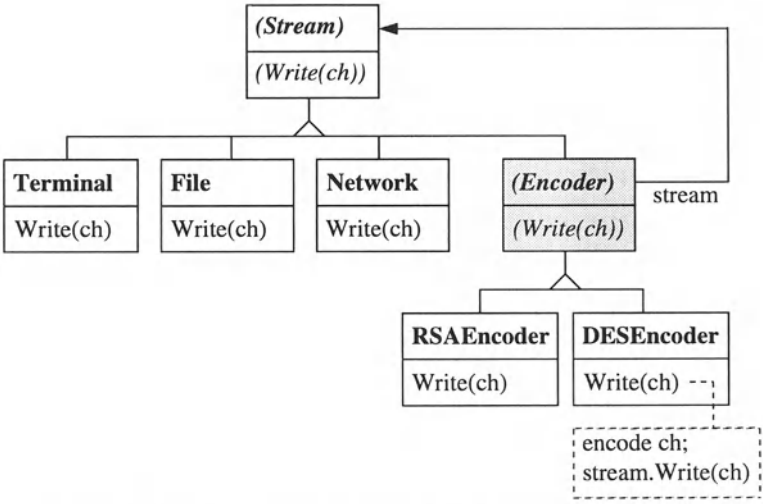


Abb. 9.15 Dekorator-Muster zur Erweiterung von *Stream* in zwei Richtungen

9.3.5 Zwilling

Zwilling

Manche Sprachen wie C++ oder Eiffel bieten mehrfache Vererbung an. Wie wir in Kapitel 5 gesehen haben, kann man damit Attribute und Methoden von mehr als einer Basisklasse erben und – was fast noch wichtiger ist – eine Unterklasse mit mehr als einer Basisklasse kompatibel machen. Allerdings bringt mehrfache Vererbung auch Probleme wie Namenskonflikte oder unübersichtliche Klassenhierarchien mit sich. Man versucht daher oft, sie zu vermeiden und mit einfacher Vererbung auszukommen. Sprachen wie Oberon-2 unterstützen mehrfache Vererbung gar nicht. Daher muß man hier immer mit einfacher Vererbung auskommen. Das *Zwillings-Muster* (*twin*) hilft einem dabei.

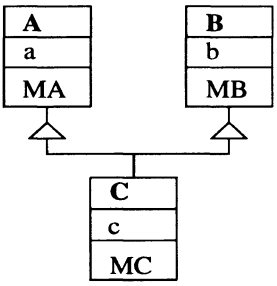


Abb. 9.16 Mehrfache Vererbung

Abb. 9.16 zeigt die Grundstruktur mehrfacher Vererbung. Eine Klasse *C* ist von zwei Klassen *A* und *B* abgeleitet und erbt deren Attribute *a* und *b* sowie deren Methoden *MA* und *MB*. *C* ist sowohl mit *A* als auch mit *B* kompatibel. Man kann *C*-Objekte also zum Beispiel in eine Liste von *A*-Objekten und in eine Liste von *B*-Objekten einhängen.

Das Zwillings-Muster beruht nun auf der Idee, die Klasse *C* in zwei Zwillingsklassen *CA* und *CB* zu zerlegen, die über Zeiger miteinander verbunden sind (Abb. 9.17). *CA* ist von *A* abgeleitet und *CB* von *B*, wodurch man mit einfacher Vererbung auskommt und Namenskonflikte vermeidet. Die Attribute und Methoden von *C* hängt man an eine der beiden Zwillingsklassen, also zum Beispiel an *CA*.

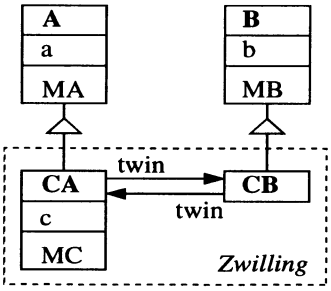


Abb. 9.17 Zwillingsklassen *CA* und *CB*

Anstatt eines *C*-Objekts, legt man ein Paar aus einem *CA*- und einem *CB*-Objekt an. Das *CA*-Objekt kann nun zum Beispiel in eine Liste von *A*-Objekten eingehängt werden, das *CB*-Objekt in eine Liste von *B*-Objekten. Somit ist die Zwillingsklasse sowohl mit *A* als auch mit *B* kompatibel, wie das auch bei mehrfacher Vererbung der Fall ist. Wenn man *MA* überschreiben möchte, so tut man das in der Klasse *CA*, wenn man *MB* überschreiben möchte, in der Klasse *CB*.

Auf die geerbten Attribute und Methoden greift man folgendermaßen zu:

```
ca.a
ca.twin.b
ca.MA
ca.twin.MB
```

Für den Zugriff auf die *B*-Komponenten muß man also eine Indirektion in Kauf nehmen. Das ist der Preis für die Vermeidung mehrfacher Vererbung, wiegt aber meist nicht allzu schwer. Beim Aufruf von *MB* kann man sich die Indirektion in der Schreibweise sogar sparen, wenn

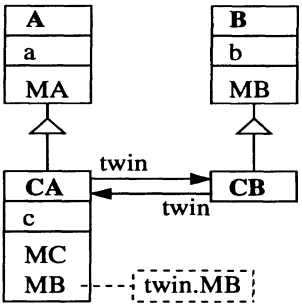


Abb. 9.18 Zwillings-Muster

man *MB* auch zu einer Methode von *CA* macht, die die Meldung an *CB* weiterleitet (Abb. 9.18).

Das folgende Beispiel zeigt das Zwillings-Muster nochmals in einer konkreten Anwendung. Nehmen wir an, ein Computerspiel besitzt Spielgegenstände wie Bälle und Schläger, die von einer gemeinsamen Basisklasse *Item* abgeleitet sind. Bälle sind *aktive* Spielgegenstände, die ständig in Bewegung sind. Solche Spielgegenstände sind von einer Basisklasse *Process* abgeleitet, wodurch sich das Klassendiagramm aus Abb. 9.19 ergibt, das mehrfache Vererbung benutzt. Bälle können sowohl in eine Liste von Spielgegenständen als auch in eine Liste aktiver Prozesse eingehängt werden.

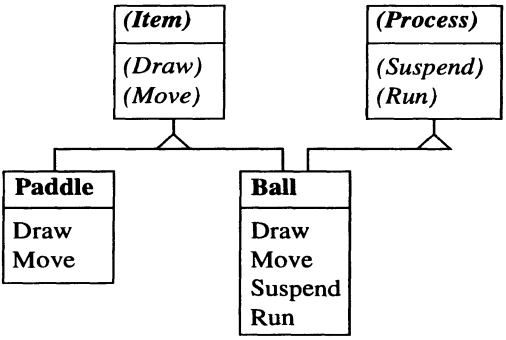


Abb. 9.19 Ballspiel mit mehrfacher Vererbung

Objekte die von *Process* abgeleitet sind, erhalten mehrmals pro Sekunde eine *Run*-Meldung. Ein Ball wird sich als Reaktion darauf ein Stück weiterbewegen. Durch die häufigen *Run*-Meldungen entsteht der Eindruck, daß der Ball sich gleichmäßig über den Bildschirm bewegt. Drückt der Benutzer eine Taste, erhalten alle Prozesse eine *Suspend*-Meldung. Ein Ball reagiert darauf durch Stillstand und Farbwechsel.

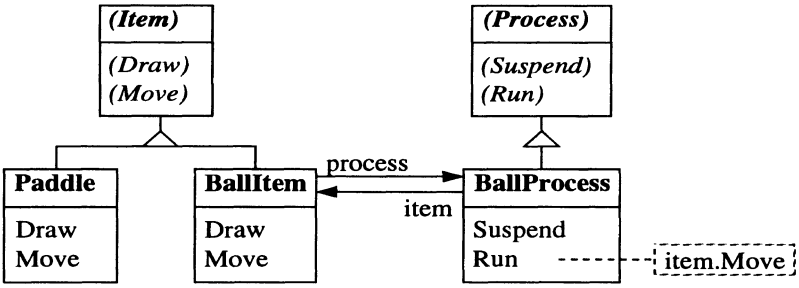


Abb. 9.20 Zerlegung der Klasse *Ball* in eine Zwillingsklasse *BallItem/BallProcess*

Modellieren wir diese Situation nun mit Hilfe des Zwillings-Musters. Die Klasse *Ball* wird in zwei Klassen *BallItem* und *BallProcess* zerlegt (Abb. 9.20). *BallItem*-Objekte hängen in der Liste der Spielgegenstände, *BallProcess*-Objekte in der Liste der aktiven Prozesse. Erhält das *BallProcess*-Objekt eine *Run*-Meldung, greift es auf sein Zwillingsobjekt *item* zu und bewegt dieses ein Stück weiter. Wird das gesamte Spielfeld neu gezeichnet, erhalten alle Objekte in der Liste der Spielgegenstände (also auch die *BallItems*) eine *Draw*-Meldung.

In diesem Beispiel ist man ohne mehrfache Vererbung ausgekommen und hat alle Anforderungen bezüglich der Kompatibilität von Bällen zu Spielgegenständen und Prozessen erfüllt.

9.4 Verhaltensmuster

Die dritte Kategorie von Entwurfsmustern umfaßt die sogenannten Verhaltensmuster. Bei diesen handelt es sich um diverse Techniken zur Lösung eines Problems in Zusammenhang mit Objekten. Wir werden in diesem Kapitel die folgenden Muster behandeln:

- Meldungsobjekt — Betrachtung einer Meldung als Objekt
- Iterator — Durchlaufen einer Objektmenge
- Beobachter — Reaktion auf eine Zustandsänderung
- Schablonenmethode — Algorithmus mit Eingriffspunkten
- Kopieren von Objekten
- Ein/Ausgabe von Objekten
- Erweiterung eines Systems zur Laufzeit

9.4.1 Meldungsobjekt

Meldungsobjekt

Methoden sind nur *eine* Möglichkeit, Meldungen zu behandeln. Eine andere Möglichkeit besteht darin, den Ausdruck "Meldung schicken" wörtlich zu nehmen. Dann ist eine Meldung ein *Datenpaket* (ein *Meldungsobjekt*), das einem anderen Objekt zur Behandlung übergeben wird. Man braucht dazu verschiedene Arten von Meldungsobjekten und eine Methode, die die Meldungsobjekte interpretiert.

Kehren wir wieder zu unserem Beispiel mit Figuren, Rechtecken und Kreisen zurück. Figuren kann man die Meldungen *Draw*, *Store* oder *Move* schicken. Wenn wir diese Meldungen als Objekte implementieren, sieht das so aus:

```
TYPE
   Message = RECORD END; (*base type of all messages*)

   DrawMsg = RECORD (Message) END;
   StoreMsg = RECORD (Message) rider: OS.Rider END;
   MoveMsg = RECORD (Message) dx, dy: INTEGER END;
```

Die konkreten Meldungsarten sind Erweiterungen der abstrakten Klasse *Message* und enthalten ihre Parameter als Recordfelder. Records dieser Art können einem sogenannten *Meldungsinterpreter* übergeben werden, der wie Abb. 9.21 eine Methode ist:

Figure
Handle (VAR m: Message)

Abb. 9.21 Klasse *Figure* mit Meldungsinterpreter

Meldungsinterpreter

Der Meldungsinterpreter *Handle* analysiert die ihm übergebenen Meldungsobjekte auf Grund ihres dynamischen Typs und reagiert auf sie. In jeder *Figurenklasse* wird er entsprechend überschrieben. Für die Klasse *Rectangle* sieht das wie in Abb. 9.22 aus:

With-Anweisung mit Varianten

Zur Interpretation der Meldung *m* wird eine With-Anweisung mit Varianten benutzt, die folgendermaßen zu lesen ist: Wenn *m* vom dynamischen Typ *DrawMsg* ist, wird die Anweisungsfolge hinter dem ersten DO-Symbol ausgeführt und *m* wird wie eine Variable mit statischem Typ *DrawMsg* behandelt; wenn *m* vom dynamischen Typ *MoveMsg* ist, wird die Anweisungsfolge hinter dem zweiten DO-Symbol ausgeführt und *m* wird wie eine Variable mit statischem Typ *MoveMsg* behandelt (daher ist der Zugriff auf *m.dx* und *m.dy* gestattet); wenn keine der Varianten zutrifft, wird der Else-Zweig ausgeführt. Wenn dieser fehlt, gibt es einen Laufzeitfehler.

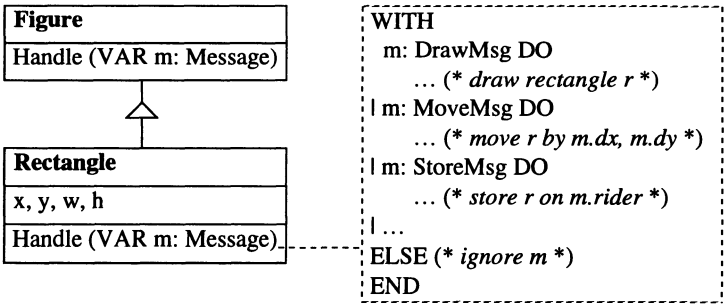


Abb. 9.22 Implementierung des Meldungsinterpreters in der Klasse *Rectangle*

Handle ignoriert in diesem Beispiel unbekannte Meldungen: der Else-Zweig der With-Anweisung ist leer. Es wäre aber auch möglich, auf unbekannte Meldungen durch eine Fehlermeldung zu reagieren oder sie an den Interpreter der Basisklasse weiterzuleiten.

Benutzung von Meldungsobjekten

Um einer Figur eine Meldung zu senden, stellt man ein passendes Meldungsobjekt zusammen und übergibt es dem Interpreter der Figur:

```
VAR f: Figure; move: MoveMsg;
...
move.dx := 10; move.dy := 20;
f.Handle(move)
```

Je nachdem, von welchem dynamischen Typ *f* ist, behandelt sein Meldungsinterpreter die *move*-Meldung unterschiedlich.

Ergebnisparameter

Ergebnisparameter einer Meldung werden im Meldungsobjekt zurückgegeben. Um zum Beispiel die Fläche einer Figur zu berechnen, kann man der Figur eine Meldung *getArea* schicken. Der Meldungsinterpreter der Figur gibt die Fläche in *getArea.value* zurück:

```
TYPE
   GetAreaMsg = RECORD (Message) value: LONGINT END;
VAR
   getArea: GetAreaMsg;
   area: LONGINT;

f.Handle(getArea);
area := getArea.value
```

Objektorientierte Programmierung mit Meldungsobjekten hat Ähnlichkeit mit der Art, wie Meldungen in *Smalltalk* behandelt werden. Auch dort wird die Meldung zur Laufzeit von einem Interpreter analysiert, der dafür sorgt, daß die entsprechende Methode aufgerufen wird. In Smalltalk ist allerdings der Meldungsinterpreter bereits ins System

eingebaut, während er in Oberon vom Programmierer implementiert wird. Auch das Ereignismodell von *Java* benutzt Meldungsobjekte.

Das Oberon-System wurde ausschließlich mit Meldungsobjekten implementiert. Auch das in Kapitel 12 beschriebene Oberon0-System benutzt Meldungsobjekte im Zusammenhang mit Bildschirmfenstern.

Vorteile

Meldungsobjekte haben gegenüber Methoden eine Reihe von Vorteilen:

- Meldungsobjekte sind *Datenpakete*. Man kann sie speichern und zu einem späteren Zeitpunkt abschicken.
- Ein Meldungsobjekt kann einer Prozedur übergeben werden, die es an mehrere (dem Sender nicht bekannte) Objekte verteilt. Man spricht in diesem Fall von einem *Broadcast*. Broadcasts sind mit Methoden nicht so einfach realisierbar, es sei denn, der Sender kennt bereits alle Empfänger und sorgt selbst dafür, daß jeder von ihnen die Meldung erhält.
- Manchmal ist es für den Sender einer Meldung einfacher, wenn er sich nicht darum kümmern muß, ob der Empfänger die Meldung versteht. Angenommen man hat eine Liste verschieder Figuren, von denen nur Rechtecke und Kreise eine *Fill*-Meldung verstehen, Linien aber nicht. Es ist einfacher (allerdings auch teurer), allen Objekten eine *Fill*-Meldung zu schicken und es den Objekten selbst zu überlassen, ob sie darauf reagieren wollen, als zu prüfen, welchen Objekten man die Meldung schicken darf. Mit Methoden ist das nicht möglich: man kann einem Objekt keine ihm unbekannte Meldung schicken, weil der Compiler prüft, ob eine entsprechende Methode in der Klasse des Empfängers existiert.
- Es ist schließlich möglich, den Meldungsinterpreter als *Prozedurvariable* anstatt als Methode zu implementieren. Dann kann man ihn zur Laufzeit austauschen und so das Verhalten eines Objekts dynamisch ändern.

Nachteile

Meldungsobjekte haben aber auch Nachteile:

- Man sieht einer Klasse nicht an, welche Meldungen man ihren Objekten schicken darf. Die verschiedenen Typen von Meldungsrecords lassen es zwar ahnen, aber diese Records müssen nicht alle im selben Modul deklariert sein. Um herauszufinden, welche Meldungen zulässig sind, muß man sich die Implementierung des Meldungsinterpreters ansehen.
- Der Interpreter analysiert die Meldungen *zur Laufzeit* mit einer With-Anweisung, deren Varianten sequentiell abgearbeitet werden. Das ist langsamer als ein Methodenaufruf, der üblicherweise durch einen direkten Zugriff auf eine Methodentabelle implementiert wird (siehe Anhang A.12.4).

- Das Senden einer Meldung ist bei Meldungsobjekten mit mehr Schreibaufwand verbunden als bei Methoden. Zuerst müssen die Eingangsparameter in das Record gepackt werden, dann wird der Meldungsinterpreter aufgerufen und anschließend kann man sich die Ergebnisparameter wieder aus dem Record holen:

```
msg.inPar := ...;
obj.Handle(msg);
... := msg.outPar
```

- Was vorhin als Vorteil gewertet wurde, kann auch ein Nachteil sein: der Compiler kann nicht prüfen, ob ein Objekt eine Meldung versteht. Folgendes Programm ist zum Beispiel für den Compiler korrekt:

```
TYPE NonsenseMsg = RECORD (Message) END;
VAR f: Figure; nonsense: NonsenseMsg;
...f.Handle(nonsense) ...
```

  Zur Laufzeit wird *f* die Meldung *nonsense* aber nicht verstehen. Das Objekt wird die Meldung ignorieren oder sogar das Programm mit einem Laufzeitfehler abbrechen. Der Fehler tritt vielleicht erst nach Monaten auf und ist dann schwer zu finden.

Meldungsobjekte haben also Vor- und Nachteile. Generell sollte man versuchen, mit Methoden zu arbeiten, weil das effizienter, sicherer und lesbarer ist. In gewissen Situationen (zum Beispiel bei Broadcasts) kann es jedoch sinnvoll sein, von der größeren Flexibilität von Meldungsobjekten Gebrauch zu machen.

9.4.2 Iterator

Iterator

Oft hat man eine Menge von Objekten vor sich, mit denen man eine bestimmte Operation ausführen möchte. Man weiß aber nicht, wie man die Objektmenge durchlaufen kann. Im Sinne der Datenabstraktion ist ihre Implementierung (Array, lineare Liste, Baum, etc.) verborgen. Ein Beispiel dafür ist etwa eine Klasse *Dictionary*, die eine Menge von Objekten der Klasse *Element* verwaltet (Abb. 9.23).

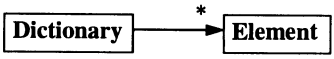


Abb. 9.23 Objektmenge mit unbekannter Implementierung

Man weiß nicht, wie *Dictionary* aufgebaut ist. Angenommen, man will alle Elemente von *Dictionary* ausdrucken. Welche Möglichkeiten gibt es dafür?

Eigene Methode für jede Operation

Die einfachste Lösung besteht darin, in *Dictionary* eine Methode *PrintAll* vorzusehen, die alle Elemente druckt:

```
PROCEDURE (VAR d: Dictionary) PrintAll;
   VAR e: Element;
BEGIN
   e := d.firstElem;
   WHILE e # NIL DO e.Print; e := e.next END
END PrintAll;
```

Die Methode *PrintAll* ist lokal zu *Dictionary* und hat daher Zugriff auf seine Implementierung. Diese Lösung ist aber unbefriedigend. Man braucht dann auch für andere Operationen eigene Methoden, zum Beispiel *StoreAll* um alle Elemente auf eine Datei abzuspeichern oder *SelectAll* um alle Elemente zu suchen, deren Schlüssel einem bestimmten Kriterium entspricht. Außerdem können diese Methoden keine Operationen benutzen, die in Unterklassen von *Element* definiert werden.

Methode zum Liefern des nächsten Elements

Eine andere Lösungsmöglichkeit besteht darin, im gleichen Modul wie *Dictionary* eine Iteratorklasse wie in Abb. 9.24 zu deklarieren.

Iterator
SetTo (dictionary) Next(): Element

Abb. 9.24 *Iterator*-Klasse

Ein *Iterator* ist ein Objekt, das man über eine Datenstruktur hinweg bewegen kann. *SetTo* setzt den Iterator an den Anfang der Datenstruktur, *Next* liefert das jeweils nächste Element daraus. Mit Hilfe eines Iterators kann man die Elemente von *Dictionary* sequentiell durchlaufen und beliebige Operationen auf sie anwenden:

```
iterator.SetTo(dictionary);
elem := iterator.Next();
WHILE elem # NIL DO
   elem.Print;
   elem := iterator.Next()
END
```

Diese Variante ist universell, erfordert aber, daß der Code zum Durchlaufen der Elemente in jedem Klienten steckt. Ein Problem ergibt sich auch, wenn die Datenstruktur zum Beispiel ein Baum ist, der am

besten rekursiv durchlaufen wird. In diesem Fall ist *Next* nicht effizient implementierbar.

Der Ergebnistyp von *Next* ist *Element*. Der tatsächliche Typ der gelieferten Objekte kann aber eine Erweiterung davon sein (z.B. *MyElement*). Durch Anwenden einer Typzusicherung kann man dem von *Next* gelieferten Objekt nun auch *MyElement*-Meldungen schicken, die in *Element* noch nicht vorgesehen sind:

```
iterator.SetTo(dictionary);
elem := iterator.Next();
WHILE elem # NIL DO
    IF elem IS MyElement THEN elem(MyElement).Store(rider) END;
    elem := iterator.Next()
END
```

Operationen als Meldungsobjekte

Als dritte Möglichkeit bietet sich an, mit Meldungsobjekten zu arbeiten. Man übergibt einem Dictionary ein Meldungsobjekt, das die gewünschte Operation ausdrückt, die man mit den Elementen ausführen möchte. Das Meldungsobjekt wird dann an alle Elemente verteilt. Jedes Element muß einen Meldungsinterpreter haben, der auf das Meldungsobjekt reagiert. Für einfache Aufgaben wie das Drucken von Elementen ist diese Lösung aber zu aufwendig.

Operationen als Prozedurvariablen

Schließlich kann man in *Dictionary* eine universelle Methode *ForAll* vorsehen, der man als Parameter eine Prozedur mitgibt. Diese Prozedur wird dann für alle Elemente aufgerufen:

```
PROCEDURE (VAR d: Dictionary) ForAll (P: PROCEDURE (e: Element));
BEGIN
    e := d.firstElem;
    WHILE e # NIL DO P(e); e := e.next END
END ForAll;
```

Der Aufruf dieser Methode sieht dann zum Beispiel so aus:

```
dictionary.ForAll(Print)
dictionary.ForAll(Store)
```

wobei *Print* und *Store* Prozeduren des Klienten sind:

```
PROCEDURE Print (e: Element);
BEGIN
    e.Print
END Print;

PROCEDURE Store (e: Element);
BEGIN
    e(MyElement).Store(rider)
END Store;
```

Man kann auf diese Weise verschiedene Prozeduren an *ForAll* übergeben und somit nahezu jede beliebige Operation mit den Elementen der Menge ausführen.

In Oberon-2 ist diese Lösung meist die einfachste und lesbarste. Einige andere Sprachen (z.B. *Sather*) besitzen spezielle Iterator-Konstrukte oder sogenannte Block-Objekte (z.B. *Smalltalk*), mit denen man Iteratoren noch bequemer implementieren kann.

9.4.3 Beobachter

Beobachter

Ein *Beobachter* (*observer*) ist ein Objekt, das am Zustand eines anderen Objekts interessiert ist. Es meldet sich bei diesem Objekt als Beobachter an und wird von ihm benachrichtigt, sobald sich der Zustand des beobachteten Objekts ändert.

Das Beobachter-Muster kommt besonders häufig bei grafischen Benutzeroberflächen vor. Abb. 9.25 zeigt zum Beispiel ein Meßwert-Objekt, dessen Zustand durch eine Zahl zwischen 0 und 100 repräsentiert wird. Zwei grafische Objekte – ein Schieber und ein Zeiger – sind als Beobachter auf den Meßwert angesetzt. Wenn sich der Meßwert ändert, werden der Schieber und der Zeiger von ihm benachrichtigt, worauf sie ihre grafische Anzeige aktualisieren.

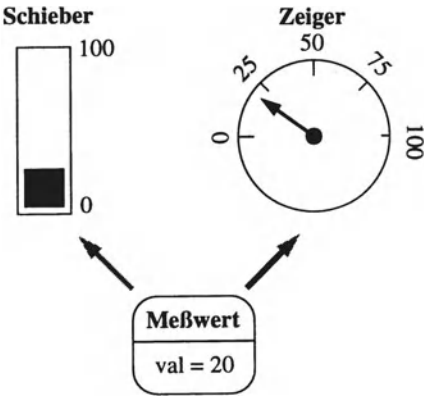


Abb. 9.25 Schieber und Zeiger als Beobachter eines Meßwerts

Das Beobachter-Muster erfüllt noch einen zweiten Zweck. Es garantiert die *Konsistenz aller Beobachter* eines Objekts. Da die Beobachter von jeder Zustandsänderung des Objekts sofort informiert werden, kennen sie zu jeder Zeit seinen aktuellen Zustand und sind daher untereinander immer konsistent.

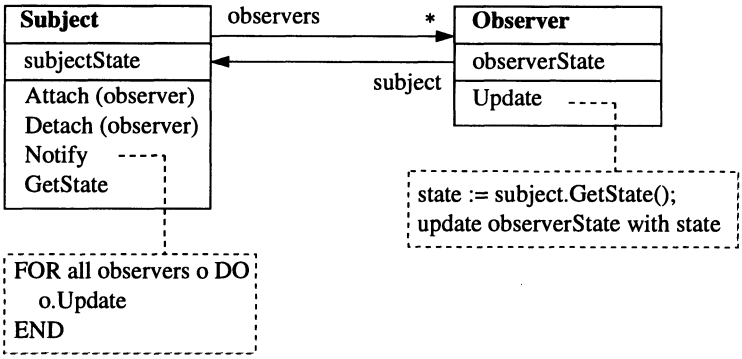


Abb. 9.26 Beobachter-Muster

Das Klassendiagramm in Abb. 9.26 zeigt nochmals das Beobachter-Muster mit seinen Klassen und Beziehungen. *Subject* (das beobachtete Objekt) schickt sich selbst ein *Notify*, sobald sich sein Zustand ändert. In der *Notify*-Methode wird dann allen registrierten Beobachtern ein *Update* geschickt. Als Reaktion darauf besorgen sich die Beobachter durch Aufruf von *GetState* den neuen Zustand von *Subject* und aktualisieren damit ihren eigenen Zustand.

Man beachte, daß sich die Beobachter bei *Subject* dynamisch an- und abmelden können (mittels *Attach* und *Detach*). Die Beziehung zwischen *Subject* und seinen Beobachtern ist also nur temporär und kann zur Laufzeit geändert werden. Es kann auch durchaus sein, daß kein einziger Beobachter angemeldet ist. Dann geht der Aufruf von *Notify* ins Leere und es wird niemand von einer eventuellen Zustandsänderung benachrichtigt.

Wenn ein Beobachter von einer Zustandänderung informiert wird, muß man ihm sagen, *was* sich geändert hat. Dafür gibt es zwei Möglichkeiten. Man kann den neuen Zustand entweder als Parameter von *Update* mitgeben (*Push-Modell*), oder man kann dem Beobachter nur mitteilen, welcher Aspekt des Zustands sich geändert hat. In diesem Fall besorgt sich der Beobachter denjenigen Teil des Zustands, der ihn interessiert, durch Aufrufe von *GetState* oder ähnlichen Methoden selbst (*Pull-Modell*). Das Push-Modell ist offensichtlich bei einem einfachen Zustand besser geeignet, das Pull-Modell bei einem komplexen Zustand.

Interaktionsdiagramm

Abb. 9.27 zeigt die Interaktionen zwischen Subject und seinen Beobachtern anhand einens sogenannten *Interaktionsdiagramms*. Die senkrechten Linien stellen Objekte dar, die waagrechten Pfeile Meldungen. Die Balken auf den Objektlinien drücken die Dauer der aufgerufenen Methoden aus. Interaktionsdiagramme wie dieses eignen sich gut, um dynamische Abläufe in Programmen zu beschreiben.

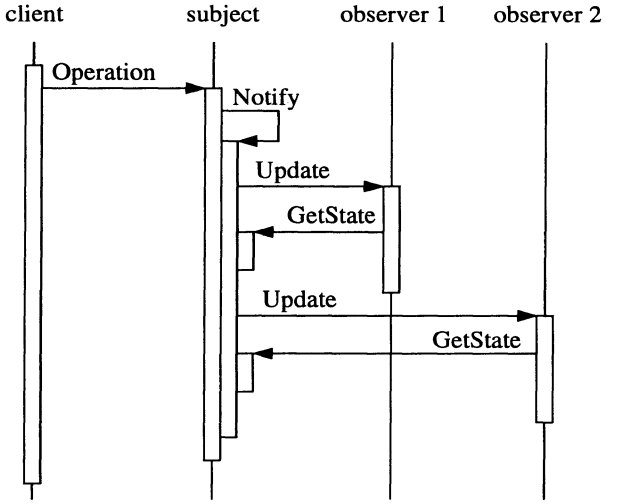


Abb. 9.27 Interaktionen im Beobachter-Muster

9.4.4 Schablonenmethode

Schablonenmethode

Eine *Schablonenmethode* (*template method*) definiert einen Algorithmus durch eine Folge von Aufrufen abstrakter Methoden. Sie legt eine Reihenfolge von Schritten fest, läßt aber die Implementierung der Schritte offen. Die Implementierung erfolgt später in Unterklassen.

Betrachten wir zum Beispiel eine Klasse *Frame* für Bildschirmfenster. Wenn ein bestimmter Bereich eines Fensters neu gezeichnet werden soll, so muß eine eventuell bestehende Selektion entfernt, ein Clipping-Bereich definiert und anschließend der Fensterinhalt neu gezeichnet werden. Diese drei Schritte sind unabhängig davon, ob es sich um ein Textfenster oder ein Grafikfenster handelt. Man kann sie in einer Schablonenmethode *Restore* zusammenfassen, wie das in Abb. 9.28 gezeigt wird.

Natürlich können die von *Restore* aufgerufenen Methoden in der abstrakten Klasse *Frame* noch nicht implementiert werden, weil sie für Text-Frames und Grafik-Frames verschieden aussehen. Ihre Implementierung erfolgt zum Beispiel in der Unterklasse *TextFrame*. Trotzdem legt *Restore* bereits die korrekte Reihenfolge ihres Aufrufs fest. Man kann nun einfach einem Text-Frame die Meldung *Restore* schicken und muß sich nicht mehr um die Aufrufe der einzelnen Teilschritte kümmern. Außerdem ist durch die Schablonenmethode sichergestellt, daß alle *Frame*-Klassen diese Teilschritte in der gleichen Reihenfolge ausführen.

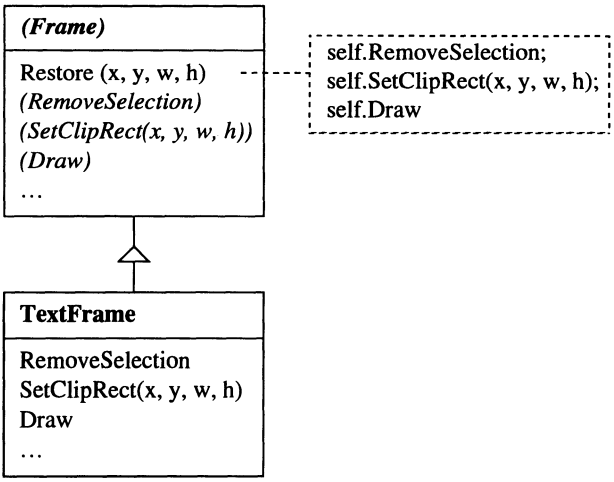


Abb. 9.28 Schablonenmethode *Restore*

Schablonenmethoden müssen nicht unbedingt nur aus abstrakten Teilschritten bestehen. Sie können bereits einen konkreten Algorithmus implementieren und nur an gewissen Stellen leere Methoden aufrufen. Durch Überschreiben dieser Methoden bekommt der Programmierer die Möglichkeit, in den Ablauf der Schablonenmethode einzugreifen. Man nennt diese leeren Methoden *Eingriffspunkte* (*hooks*), weil sie die Möglichkeit bieten, sich in einen bestehenden Algorithmus mit eigenem Code einzuhängen.

Eingriffspunkte in Algorithmen

Betrachten wir dazu wieder ein Beispiel. In einem Grafikfenster gibt es eine Methode *TrackMouse*, die die Mausbewegungen verfolgt und den Mauszeiger zeichnet. Wenn man die Bewegungen des Mauszeigers

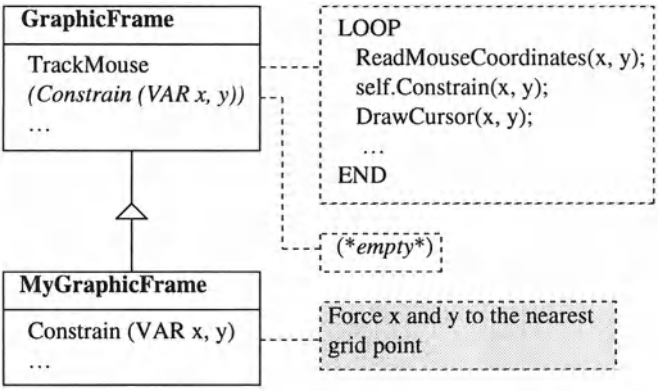


Abb. 9.29 *Constrain* als Eingriffspunkt in die Methode *TrackMouse*

auf ein festes Gitter beschränken möchte, kann man in *TrackMouse* eine leere Methode *Constrain* aufrufen, was zunächst ohne Wirkung ist. Der Programmierer kann *Constrain* aber in einer Unterklasse so überschreiben, daß die gelesenen Mauskoordinaten auf den nächstgelegenen Gitterpunkt gezwungen werden (Abb. 9.29).

Es ist eine gängige Technik, einen Algorithmus durch möglichst viele Eingriffspunkte flexibel zu machen. Wenn man allerdings zu viele Eingriffspunkte vorsieht, leidet die Effizienz des Algorithmus. Die Klassenschnittstelle wird dann auch durch eine Fülle leerer Methoden unübersichtlich.

9.4.5 Kopieren von Objekten

Kopieren von Objekten

Objekte zu kopieren scheint trivial zu sein. Wenn man allerdings den dynamischen Typ des zu kopierenden Objekts nicht kennt, ist diese Aufgabe gar nicht so einfach. Wie kann man vorgehen?

Eine naheliegende Lösung besteht darin, dem Objekt eine *Copy*-Meldung zu schicken. Durch die dynamische Bindung wird die *Copy*-Methode des dynamischen Typs des Objekts aufgerufen. Dort wird ein neues Objekt des Empfängertyps angelegt und mit den Attributwerten des Originals gefüllt. Diese scheinbar einfache Lösung hat aber ihre Tücken, wie Abb. 9.30 zeigt.

Die *Copy*-Methode von *Circle* muß nicht nur ihre eigenen Attribute kopieren, sondern auch die der Basisklasse *Figure*. Das kann sie nur, wenn diese Attribute exportiert sind. Sie kann auf keinen Fall zum Kopieren der Basisattribute die *Copy*-Methode von *Figure* verwenden, weil diese Methode ein neues Objekt erzeugt anstatt die Basisattribute in das Objekt *c* zu kopieren.

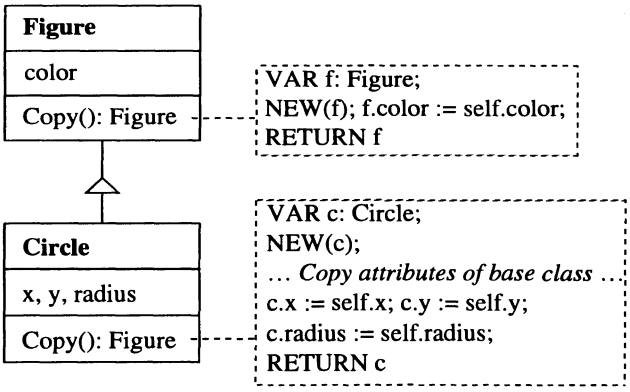


Abb. 9.30 Probleme beim Kopieren von Objekten

Man kann sich hier mit einem Trick behelfen. Man gibt *Copy* das Objekt, in das die Attribute kopiert werden sollen, als Var-Parameter mit. Beim ersten Aufruf gibt man NIL mit, wodurch die aufgerufene Methode weiß, daß sie zuerst ein neues Objekt anlegen muß, bevor sie die Attribute kopiert. Dies ist in Abb. 9.31 gezeigt.

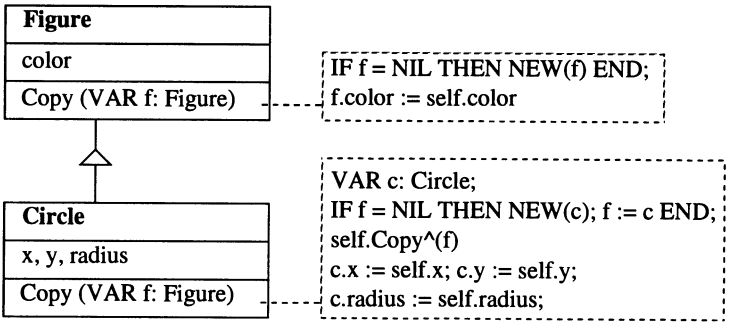


Abb. 9.31 Kopieren von Objekten

Um eine Kopie *f* eines Objekts *g* zu erzeugen, schreibt man

```
f := NIL; g.Copy(f)
```

Wenn *g* den dynamischen Typ *Circle* hat, wird die *Copy*-Methode von *Circle* aufgerufen. Weil *f* dort den Wert NIL hat, wird ein neues *Circle*-Objekt angelegt. Beim Aufruf von *Copy* aus der Basisklasse ist *f* nicht mehr NIL, daher wird kein neues Objekt mehr angelegt, sondern es wird nur das Attribut *color* kopiert.

Modul Types

Natürlich ist es unschön, daß man den Parameter von *Copy* vor dem Aufruf auf NIL setzen muß. Man kann sich das sparen, wenn einem das Laufzeitsystem Operationen zum Arbeiten mit Typen und zum Erzeugen von Objekten dieser Typen zur Verfügung stellt. Das Oberon-System bietet solche Operationen in einem Modul namens *Types* an (Anhang B.4). Die wichtigsten Teile dieses Moduls sind:

```
DEFINITION Types;
   IMPORT SYSTEM, Modules;
   TYPE
      Type = POINTER TO RECORD
         name: ARRAY 32 OF CHAR;   (* type name*)
         module: Modules.Module;   (* module descriptor *)
      END;
   PROCEDURE TypeOf (obj: SYSTEM.PTR): Type;
   PROCEDURE NewObj (t: Type; VAR obj: SYSTEM.PTR);
   PROCEDURE This (m: Modules.Module; name: ARRAY OF CHAR): Type;
END Types.
```

Type ist ein *Typdeskriptor*, beschreibt also gewisse Eigenschaften eines Typs wie seinen Namen oder das Modul, in dem er deklariert ist. Wenn *p* ein Zeiger auf ein Record vom Typ *T* ist, liefert *TypeOf(p)* den Typdeskriptor von *T* (SYSTEM.PTR ist ein Typ, der mit jedem Zeigertyp kompatibel ist). *NewObj(t, obj)* erzeugt ein neues Objekt *obj* von demjenigen Typ, der durch den Typdeskriptor *t* beschrieben wird. *This(m, name)* liefert den Typdeskriptor des Typs namens *name* aus dem Modul *m*.

Mit Hilfe des Moduls *Types* kann man *Copy* nun eleganter implementieren. Dies wird in Abb. 9.32 gezeigt.

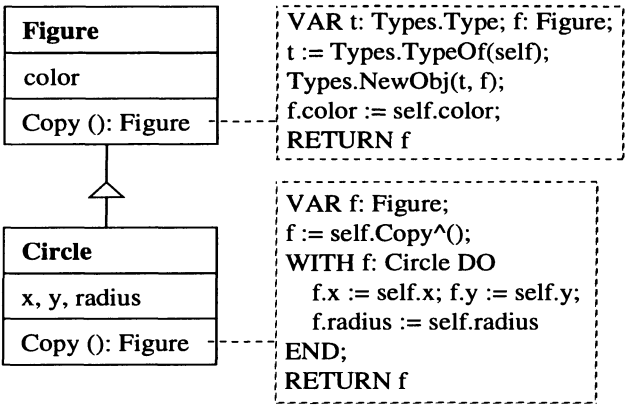


Abb. 9.32 Kopieren von Objekten mit Hilfe des Moduls *Types*

Um eine Kopie eines Objekts *g* zu erzeugen, schreibt man nun

```
f := g.Copy()
```

Wenn *g* den dynamische Typ *Circle* hat, wird die *Copy*-Methode von *Circle* aufgerufen. Diese ruft gleich die *Copy*-Methode der Basisklasse auf. Mit Hilfe des Moduls *Types* wird dort ein neues Objekt *f* vom gleichen dynamischen Typ wie der Empfänger erzeugt, also ein *Circle*-Objekt. Nachdem das Attribut *color* kopiert wurde, wird *f* an die *Copy*-Methode von *Circle* zurückgegeben. Dort werden die restlichen Attribute kopiert. Die With-Anweisung wandelt den statischen Typ von *f* in *Circle* um, sodaß beim Kopieren auf die Attribute *f.x*, *f.y* und *f.radius* zugegriffen werden kann.

9.4.6 Ein/Ausgabe von Objekten

Ein/Ausgabe

In fast allen Programmen muß man irgendwann Objekte auf eine Datei schreiben und sie später wieder einlesen. Dabei stellt sich ein ähnliches Problem wie beim Kopieren von Objekten: Wie kann man Objekte schreiben und lesen, deren dynamischen Typ man nicht kennt?

Das Schreiben solcher Objekte ist noch nicht besonders schwierig. Man schickt einem Objekt einfach eine *Store*-Meldung und vertraut darauf, daß das Objekt selbst weiß, welche Attribute es auf die Datei schreiben muß. Das Lesen der Objekte ist wesentlich problematischer. Bevor ein Objekt gelesen werden kann, muß es erst erzeugt werden. Wie weiß man aber, von welchem Typ das zu erzeugende Objekt sein soll? Man kennt ja seinen dynamischen Typ nicht.

Die Lösung besteht darin, neben den *Attributen* eines Objekts auch seinen *Typnamen* auf die Datei zu schreiben. Unter Verwendung des Moduls *Types* aus Kapitel 9.4.5 kann man aus jedem Objekt seinen Typnamen gewinnen und umgekehrt aus jedem Typnamen ein Objekt dieses Typs erzeugen. Die beiden folgenden Prozeduren *StoreObj* und *LoadObj* schreiben ein beliebiges Objekt samt Typnamen auf eine Datei und lesen es wieder ein.

```
PROCEDURE StoreObj (VAR r: OS.Rider; obj: Object);
   VAR type: Types.Type;
BEGIN
   IF obj = NIL THEN r.WriteString("");
   ELSE
      type := Types.TypeOf(obj);
      r.WriteString(type.module.name);
      r.WriteString(type.name);
      obj.Store(r)
   END
END StoreObj;

PROCEDURE LoadObj (VAR r: OS.Rider; VAR obj: Object);
   VAR mod: Modules.Module; type: Types.Type;
      modName, typeName: ARRAY 32 OF CHAR;
BEGIN
   r.ReadString(modName);
   IF modName = "" THEN obj := NIL
   ELSE
      r.ReadString(typeName);
      mod := Modules.This(modName);
      type := Types.This(mod, typeName);
      Types.NewObj(type, obj);
      obj.Load(r)
   END
END LoadObj;
```

Die beiden Prozeduren gehen davon aus, daß alle zu schreibenden und zu lesenden Objekte von einer Klasse *Object* abgeleitet sind, die die beiden Meldungen *Load* und *Store* unterstützt. Der Typ *OS.Rider* repräsentiert eine Lese/Schreibposition in einer Datei (siehe Anhang B.5). *StoreObj* holt sich den Typdeskriptor von *obj* und schreibt dessen Modulnamen (*type.module.name*) sowie dessen Typnamen (*type.name*) auf die Datei. *LoadObj* liest den Modulnamen und den Typnamen, holt sich den Moduldeskriptor *mod* und den Typdeskriptor *type* und erzeugt dann mit *NewObj* ein neues Objekt dieses Typs.

Während *StoreObj* im Prinzip zu einer Methode von *Object* gemacht werden könnte ist das bei *LoadObj* nicht möglich, da man ja einem noch nicht existierenden Objekt keine Meldung schicken kann. Aus Symmetriegründen sind daher sowohl *LoadObj* als auch *StoreObj* als gewöhnliche Prozeduren implementiert.

Abb. 9.33 zeigt, wie die *Load*- und *Store*-Methoden einer Klasse *A* und ihrer Unterklasse *B* implementiert werden, so daß alle ihre Attribute korrekt gelesen und geschrieben werden. Man beachte, daß *B* ein Attribut *b* hat, das selbst wieder ein Objekt ist. Daher wird dieses Attribut mit *StoreObj* ausgegeben und mit *LoadObj* eingelesen. Beim Einlesen muß eine Typzusicherung verwendet werden, weil *LoadObj* einen Parameter vom statischen Typ *Object* liefert.

Wenn das Laufzeitsystem kein Operationen zur Umwandlung eines Objekts in seinen Typnamen und umgekehrt zur Verfügung stellt, kann man sich folgendermaßen behelfen: Jedes Objekt muß dann eine

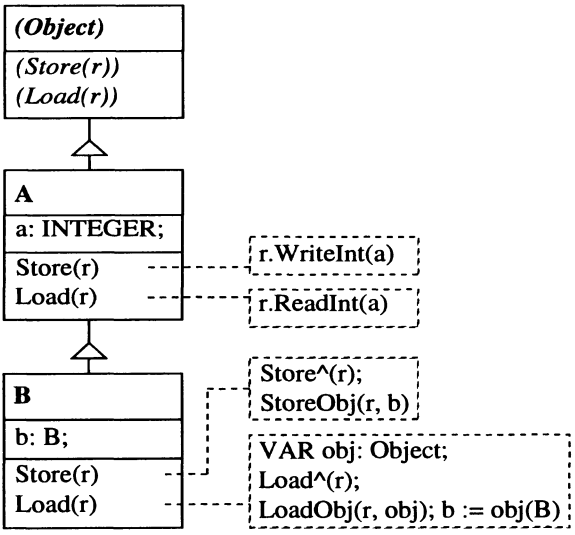


Abb. 9.33 Implementierung der *Load*- und *Store*-Methoden

Meldung *GetTypeName* unterstützen, die den Typnamen dieses Objekts liefert. Ferner muß man sich eine Tabelle anlegen, in der für jeden verwendeten Typ ein Prototypobjekt dieses Typs samt Typnamen abgelegt ist. Benötigt man nun ein Objekt eines bestimmten Typnamens, sucht man sich in der Tabelle das entsprechende Prototypobjekt und erzeugt eine Kopie. Natürlich ist das für den Programmierer mit mehr Aufwand verbunden als die oben beschriebene Technik.

Persistente Objekte

Das Schreiben und Lesen von Objekten ist die Grundlage zur Implementierung persistenter Objekte. Ein Objekt heißt *persistent*, wenn es das Programm überlebt, das es erzeugt hat. Persistente Objekte werden in Datenbank-ähnlichen Anwendungen verwendet. Meist sind sie mit anderen Objekten zu einem Graphen-ähnlichen Geflecht verbunden. Beim Lesen und Schreiben eines solchen Geflechts muß man darauf achten, daß jedes Objekt nur einmal rausgeschrieben wird, auch wenn es mehrere Zeiger auf das Objekt gibt. Entsprechende Techniken sind Büchern über Graphenalgorithmen zu entnehmen.

9.4.7 Erweiterung eines Systems zur Laufzeit

Dynamische Erweiterungen

In Kapitel 8.3 haben wir gesehen, daß man einen Grafikeditor zur Laufzeit um neue Objekte (Rechtecke, Kreise, Linien) erweitern kann, die zum Zeitpunkt seiner Implementierung noch nicht bekannt waren. In diesem Kapitel wollen wir uns ansehen, wie man das in Oberon macht, ohne das zu erweiternde Programm zu entladen, neu zu binden und wieder zu laden.

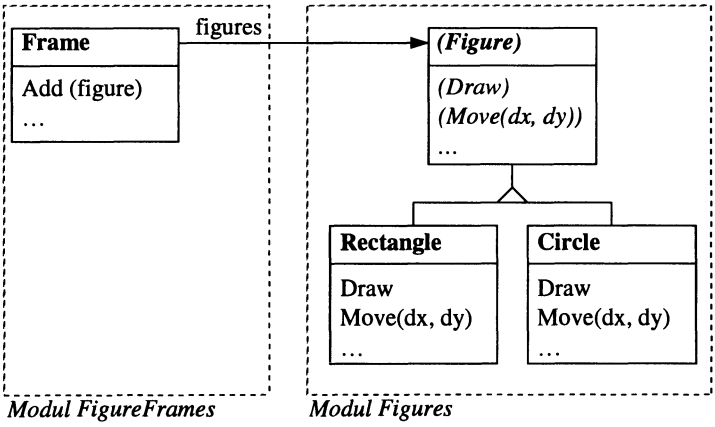


Abb. 9.34 Figuren-Familie in einem Grafikeditor

Rufen wir uns noch einmal das Beispiel aus Kapitel 8.3 in Erinnerung: Der Grafikeditor arbeitet dort mit einer Figuren-Familie bestehend aus Rechtecken, Kreisen oder anderen Figuren (Abb. 9.34). Wir nehmen an, daß alle Klassen dieser Familie in einem Modul *Figures* deklariert sind. Zum Editor gehört auch ein Modul *FigureFrames* mit einer Klasse *Frame*, die dafür verantwortlich ist, die Figuren zu verwalten und auf dem Bildschirm darzustellen. Mit der Methode *Add* kann man neue Figuren hinzufügen.

Erweiterung eines Grafikeditors

Das ist der Kern des Editors. Bei seiner Implementierung muß man noch nicht wissen, welche Figurenarten es später einmal geben wird. Der Editor kann mit jeder beliebigen Unterklasse von *Figure* arbeiten.

Nun möchte man den Editor um Ellipsen erweitern. Was ist zu tun:

Einführen von Ellipsen

1. Man definiert eine Klasse *Ellipse* als Unterklasse von *Figure* und überschreibt die geerbten Methoden (Abb. 9.35).
2. Man implementiert ein Kommando *New*, das ein Ellipsen-Objekt erzeugt und in die Liste der anderen Figuren im aktuellen Frame einfügt.

```
PROCEDURE New;
   VAR e: Ellipse;
BEGIN
   NEW(e);
   ... (*fill e.x, e.y, e.a, and e.b*) ...
   FigureFrames.currentFrame.Add(e)
END New;
```

Die Klasse *Ellipse* und das Kommando *New* verpackt man in ein neues Modul *Ellipses*. Die bestehenden Module des Editors müssen nicht angerührt werden. Um ein neues Ellipsen-Objekt im Editorfenster zu

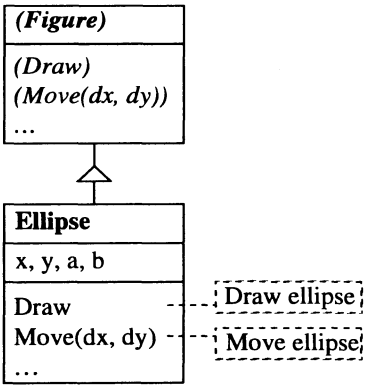


Abb. 9.35 Ableitung einer neuen Unterklasse *Ellipse* aus *Figure*

installieren, ruft man das Kommando *Ellipses.New* auf. Dabei geschieht folgendes:

1. Falls das Modul *Ellipses* noch nicht geladen war, wird es jetzt geladen und zum Editor hinzugefügt. Dadurch wird der Editor zur Laufzeit um ein neues Modul erweitert.
2. Das Kommando *New* wird ausgeführt. Es erzeugt ein Ellipsen-Objekt und fügt es in die Figurenliste des aktuellen Frames ein.
3. Der Frame schickt der neu eingefügten Figur (deren Typ er nicht kennt) eine *Draw*-Meldung, die bewirkt, daß die Ellipse gezeichnet wird.

Dynamisches Laden des Moduls Ellipses

Man beachte, daß das Modul *Ellipses* erst bei Bedarf geladen und zum bereits laufenden Editorkern gebunden wird. Weder *Figures* noch *FigureFrames* kennen (d.h. importieren) *Ellipses*. Sie können daher übersetzt und benutzt werden, lange bevor *Ellipses* existiert. Umgekehrt kennt und importiert *Ellipses* die Module *Figures* und *FigureFrames*. *Ellipses* baut auf ihnen auf und erweitert sie.

Der Editorkern kann das unbekannte Modul *Ellipses* aufgrund der dynamischen Bindung benutzen. Er sieht im Ellipsen-Objekt ein Exemplar der abstrakten Klasse *Figure* und kommuniziert mit ihm über Meldungen, die zum Aufruf von Methoden aus dem in der Import-Hierarchie höher gelegenen Modul *Ellipses* führen. Man nennt solche Aufrufe daher im Englischen *Up-calls*.

Die Möglichkeit, ein System zur Laufzeit zu erweitern, ohne es vorher zu entladen, neu zu compilieren und zu binden, macht die objektorientierte Programmierung erst so richtig mächtig. Das Oberon-System bietet durch Kommandos und dynamisches Nachladen von Modulen die Voraussetzungen für diese Art von Erweiterbarkeit.

10 Objektorientierter Entwurf

Ein Programm zu entwerfen, heißt, es in kleinere, überschaubare Teile zu zerlegen und deren Wechselwirkungen zu beschreiben. Die Teile können Module, Prozeduren, Datenstrukturen oder Klassen sein. Beim objektorientierten Entwurf interessieren uns vor allem Klassen. Unsere Frage lautet daher: Wie findet man die zur Implementierung eines Systems benötigten Klassen?

10.1 Aufgabenorientierte Sicht

Herkömmlicher Programmentwurf beginnt mit der Frage: Was soll das Programm leisten? Man orientiert sich also an den *Aufgaben*, die zu lösen sind. Dabei geht man von der Gesamtaufgabe aus, zerlegt sie in mehrere Teilaufgaben und jede Teilaufgabe wieder in kleinere Teilaufgaben, bis diese so einfach sind, daß man sie direkt in einer Programmiersprache formulieren kann.

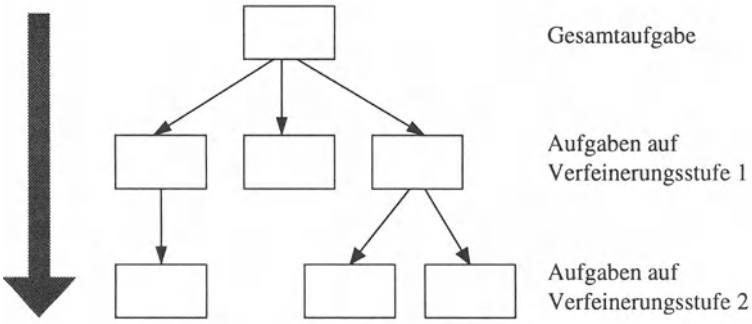


Abb. 10.1 Programmhierarchie beim aufgabenorientierten Entwurf. Pfeile deuten die Benutzt-Beziehung an.

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

Schrittweise Verfeinerung

Diese Vorgehensweise nennt man *schrittweise Verfeinerung* [Wir71]. Man arbeitet sich vom Abstrakten zum Konkreten vor, von der Gesamtaufgabe zu den Details. Die schrittweise Verfeinerung ist eine *Top-down-Methode*. Sie führt zu einer Hierarchie von Bausteinen, wie sie in Abb. 10.1 gezeigt ist.

Die schrittweise Verfeinerung hat viele Vorteile: sie läßt sich einfach und systematisch anwenden und führt zu sauber gegliederten Programmen. Wichtige Aufgaben (z.B. die Steuerlogik) werden als erstes entworfen, weniger wichtige Details zum Schluß; der Entwurf der Steuerlogik formt den ganzen restlichen Entwurf.

Sie hat aber auch Nachteile: Gerade die Steuerlogik ist der heikelste Teil eines Programms. Am Anfang weiß man oft noch gar nicht genau, wie sie aussehen soll, ja es ist manchmal noch nicht einmal klar, ob es nur ein einziges Hauptprogramm geben soll oder mehrere Programme, die gleichberechtigt nebeneinander stehen. Bei einem Betriebssystem kann man zum Beispiel schwer sagen, welcher Teil davon das Hauptprogramm ist. Wo soll die Verfeinerung also beginnen? Natürlich kann man jeden Teil für sich verfeinern, aber das führt zu getrennten Programmhierarchien ohne gemeinsame Teile an der Basis.

Schrittweise Verfeinerung fördert die Wiederverwendung von Software nicht. Alle Teilaufgaben sind auf die Bedürfnisse der übergeordneten Aufgabe zugeschnitten, daher ist das entstehende Programm ein Stück Maßarbeit. Seine Teile sind kaum in anderen Programmen wiederverwendbar.

Schließlich sind schrittweise verfeinerte Programme gegenüber Änderungen empfindlich. Wenn sich die Anforderungen an das Hauptprogramm ändern, muß die Zerlegung oft ganz anders erfolgen, was große Teile des Programmentwurfs über den Haufen werfen kann.

Schrittweise Verfeinerung ist also eine Technik, die sich zwar hervorragend für den Entwurf kleiner Programme oder Algorithmen eignet, aber weniger gut für den Entwurf großer Systeme.

10.2 Objektorientierte Sicht

Beim objektorientierten Entwurf lautet die Hauptfrage nicht: *Was* macht das System? Die Frage lautet vielmehr: *Womit* macht das System etwas? Man fragt also nach den zentralen Daten und den auf sie anwendbaren Operationen, also nach den Objekten. Da diese Objekte kaum als die Spitze des Systems betrachtet werden können, ist objektorientierter Entwurf eher eine *Bottom-up-Technik*.

Man formt die zentralen Daten eines Systems zu Objekten, die mittels festgelegter Operationen wie kleine autonome Maschinen bedient

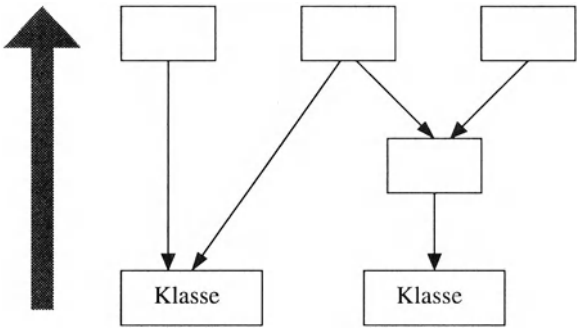


Abb. 10.2 Programmhierarchie beim objektorientierten Entwurf. Pfeile deuten die Benutzt-Beziehung an.

werden können. Darauf aufbauend läßt sich dann die Steuerlogik formulieren.

Durch diese Vorgehensweise erhält man eine Menge von Klassen, die dem System als Basis zugrunde liegen. Darauf können andere Klassen, Prozeduren und Module aufsetzen, so daß ein System entsteht, das mehrere gleichberechtigte Programmteile an der Spitze haben kann (Abb. 10.2).

Vorteile und Nachteile

Objektorientierter Entwurf hat unter anderem folgende Vorteile:

- Die entstehenden Klassen sind meist ein Abbild der Dinge in der realen Welt; Programme werden dadurch problemnäher und besser verständlich.
- Objektorientierter Entwurf läßt mehrere gleichberechtigte Programmteile an der Spitze eines Systems zu.
- Die Klassen an der Basis wurden nicht auf bestimmte Klienten zugeschnitten. Sie lassen sich daher oft auch in anderen Programmen wiederverwenden.

Ein Nachteil des objektorientierten Entwurfs ist, daß die Klassen an der Basis oft zu allgemein sind. Wenn man Binärbäume braucht, in denen man nur Elemente einfügen und suchen will, sind Operationen wie Abspeichern, Einlesen oder Mischen von Bäumen überflüssig. Man muß sie mitschleppen, obwohl man sie gar nicht braucht. Das ist der Preis, den man für die Wiederverwendung von Software zahlen muß.

Top-down versus bottom-up

Objektorientierter Entwurf ist also eher eine Bottom-up-Technik. Das heißt aber nicht, daß man Programme *ausschließlich* bottom-up entwerfen sollte. In der Praxis geht man vielmehr sowohl bottom-up als auch top-down vor, je nachdem, welche Technik gerade die geeig-

netere ist. Wenn man zum Beispiel Software zur Steuerung eines Computernetzes entwirft, kümmert man sich zuerst um die Klassen an der Basis, die das Netzwerk, seine Schichten und Protokolle modellieren. Anschließend verwendet man diese Klassen, um die Steuerlogik top-down zu entwerfen. Vielleicht kommt man dabei an einen Punkt, wo man merkt, daß man ein Namensverzeichnis für Netzwerkadressen braucht. Also geht man wieder zurück und entwirft eine Klasse an der Basis, die diese Dienste anbietet. Schließlich setzt man den Top-down-Entwurf wieder fort.

Manche Leute empfehlen sogar, beim objektorientierten Entwurf weder top-down noch bottom-up vorzugehen, sondern vom Vertrauten zum weniger Vertrauten [Bud91].

10.3 Wie findet man Klassen?

Anforderungsdefinition

Ausgangspunkt für den Klassenentwurf ist eine *Anforderungsdefinition* (*Pflichtenheft*), die festlegt, was das Programm leisten soll. Sie beschreibt noch keine Programmstruktur, daher muß man noch nicht auf Klassen, Methoden und andere objektorientierte Aspekte achten. Jede beliebige, dem Leser geläufige Notation ist dafür geeignet (z.B. Text, Datenflußdiagramme, etc.).

Ausgehend von der Anforderungsdefinition versucht man, Klassen zu finden. Obwohl es naiv wäre zu glauben, man könne durch blindes Befolgen von Regeln zu guten Entwürfen gelangen, hört man immer wieder die Frage: Wie findet man die Klassen zu einem System? Man kann kein mechanisches Verfahren dafür angeben, denn Entwurf erfordert Erfahrung und Fingerspitzengefühl. Alles was man tun kann ist, die wichtigsten Überlegungen zu nennen, von denen man sich leiten lassen sollte.

10.3.1 Grundsätzliche Entwurfsüberlegungen

Objekte der realen Welt

Programme werden verständlicher, wenn man sie so zerlegt, daß Klassen den Dingen entsprechen, die in der Realität vorkommen: Wenn man zum Beispiel einen Editor entwirft, mit dem man Texte in Bildschirmfenstern bearbeiten möchte, liegt es nahe, Texte auf eine Klasse *Text* abzubilden und Fenster auf eine Klasse *Window*. Am besten beginnt man den Entwurf mit folgenden drei Fragen:

Klassen

1. *Was sind die physischen und logischen Objekte des Systems in der realen Welt?* Diese Frage führt zu den Klassen. Physische Ob-

jekte sind zum Beispiel Schalter, Sensoren oder Anzeigen. Logische Objekte sind etwa ein Prozeß, eine Meßreihe oder ein Kommando. Alle Begriffe, die von zentraler Bedeutung für das System sind, wichtige Daten repräsentieren oder verschiedene Zustände annehmen können, sind Kandidaten für Klassen.

2. *Welche Operationen kann man mit diesen Objekten ausführen?* Diese Frage führt zu den Methoden. Die Operationen einer Klasse *Sensor* sind zum Beispiel Einschalten, Ausschalten, Ablesen, usw. Bei der Suche nach Methoden sollte man sich nicht zu sehr von den momentanen Anforderungen einengen lassen, sondern die Wiederverwendbarkeit der Klasse ins Auge fassen. Jede Operation, die für eine Klasse vernünftig ist, ist Kandidat für eine Methode. *Methoden*

   Es hat sich als nützlich erwiesen, verschiedene Szenarien durchzuspielen, in denen die Klasse vorkommt, und sich dabei zu fragen: Welche Ereignisse können eintreten? Welche Objekte sollen auf diese Ereignisse reagieren und *wie* sollen sie reagieren? Welche anderen Aktionen oder Ereignisse werden dadurch ausgelöst?

3. *Welche Daten müssen in einem Objekt gespeichert werden, damit die Operationen ihre Aufgabe erfüllen können?* Diese Frage führt zu den Attributen einer Klasse. Die Attribute stellen den Zustand eines Objekts dar, der über Methoden verändert und abgefragt werden kann. Sie sind die konkrete Datenstruktur, die vor den Klienten der Klasse verborgen wird. Die Attribute einer Klasse *Sensor* sind zum Beispiel sein momentaner Meßwert und ein Abtastintervall. *Attribute*

Durch diese drei Fragen kann man bereits die Klassen mit all ihren Bestandteilen finden und ihre Schnittstellen spezifizieren. Eine Klasse *Sensor* könnte etwa wie in Abb. 10.3 aussehen.

Was hat man bis hierher geleistet? Man hat mit jeder Klasse ein Stück *Verhalten* aus dem Programm herausgezogen und an einer Stelle konzentriert. Das verbleibende Programm ist "ausgedünnt" und somit

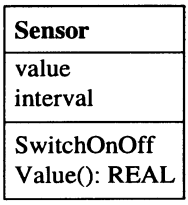


Abb. 10.3 Klasse *Sensor*

weniger komplex. Es kann mit den Klassen auf höherer Ebene weiterarbeiten. Wenn man genügend viele Klassen auf diese Weise gewonnen hat, ist die verbleibende Steuerlogik oft nicht mehr allzu kompliziert.

Es kann allerdings auch vorkommen, daß man *zu viele* Klassen bildet. Dann ist das Programm nicht mehr aufgrund seiner Steuerlogik komplex, sondern aufgrund seiner vielen kleinen Bausteine und ihrer Beziehungen.

10.3.2 Zusätzliche Entwurfsüberlegungen

Neben den drei Grundfragen nach den Klassen, Methoden und Attributen sind noch folgende Überlegungen nützlich:

Typische Muster

1. *Gibt es Situationen, die sich mit Klassen besonders elegant lösen lassen?* Wo kommen Bausteine in Form von Varianten vor? Wo möchte man später neue Varianten hinzufügen? Wo möchte man einen Baustein zur Laufzeit gegen einen anderen auswechseln? Situationen dieser Art wurden in den Kapiteln 8 und 9 beschrieben. Bei ihnen lohnt sich der Einsatz von Klassen ganz besonders, weil man Erweiterbarkeit und dynamische Bindung ausnützen kann.

Datenabstraktion

2. *Welche systemabhängigen Teile sollen vor Klienten verborgen werden?* Programme enthalten oft systemnahe Details, die man schwer versteht und die das erste sind, was bei einer Portierung geändert werden muß. Es ist ratsam, solche Details in Modulen oder in Klassen zu kapseln, so daß Änderungen an ihnen lokal bleiben und Klienten sie benutzen können, ohne ihre Implementierung zu kennen.

Vorhersehbare Änderungen

3. *Welche Teile der Software werden sich möglicherweise ändern? Wie kann man die Auswirkungen solcher Änderungen gering halten?* Oft weiß man schon vor der Fertigstellung eines Programms, daß Teile davon später einmal geändert werden müssen, weil man etwa nach einer effizienteren Lösung sucht, weil man das Programm portieren möchte oder weil sich äußere Bedingungen ändern. Solche Teile sollte man in einer Klasse verbergen und mit einer Schnittstelle versehen, die gegenüber Änderungen robust ist.

Basisdienste

4. *Welche gleichartigen Basisdienste werden auch in anderen Programmen benötigt?* Es gibt Untersuchungen, die besagen, daß mehr als die Hälfte aller Aufgaben eines Programms in ähnlicher

Form auch in anderen Programmen vorkommen. Beispiele sind die Listenverwaltung, Text- und Grafikoperationen sowie die Ein- und Ausgabe von Texten. Es ist sinnvoll, solche Basisdienste vom Programm, das sie benutzt, zu entkoppeln, damit sie als unabhängige Bausteine wiederverwendet werden können.

Wie machen es andere

5. *Welche Zerlegung ist aus ähnlichen Systemen bekannt?* Wie so vieles lernt man guten Entwurf weniger durch Regeln als durch Erfahrung. Editoren sind immer ähnlich aufgebaut, ebenso Buchhaltungs- oder Simulationsprogramme. Durch das Studium vorhandener Systeme sammelt man ein Repertoire von Entwurfsmustern und lernt abzuschätzen, wann ihr Einsatz sinnvoll ist. In diesem Sinne sei dem Leser Kapitel 12 dieses Buches empfohlen, das die vollständige Implementierung eines Fenstersystems mit einem erweiterbarem Text- und Grafikeditor enthält.

10.3.3 Ableitung von Klassen aus einem Text

Methode von Abbott

Abbott schlägt ein Verfahren vor, mit dem man Klassen fast mechanisch aus einer verbalen Aufgabenbeschreibung ableiten kann [Abb83]. Er rät, auf die im Text vorkommenden Substantive, Verben und Adjektive zu achten und daraus die Klassen, Methoden und Attribute abzuleiten.

Die *Substantive* des Textes sind Kandidaten für Klassen oder Attribute. Sie beschreiben die Objekte, mit denen etwas ausgeführt wird oder die Eigenschaften eines Objekts. Die *Verben* des Textes sind Kandidaten für Methoden. Sie beschreiben die Operationen, die mit den Objekten ausgeführt werden. Die *Adjektive* des Textes schließlich deuten wieder auf Attribute hin. Sie halten eine Eigenschaft oder einen Zustand eines Objekts fest.

Wenn eine Spezifikation den Satz enthält: "Der Editor muß Figuren zeichnen und löschen sowie ihre Größe verändern können", so sind *Editor*, *Figur* und *Größe* die darin vorkommenden Substantive. *Editor* und *Figur* sind zentrale Objekte und deuten auf Klassen hin. *Größe* bezeichnet aber lediglich eine Eigenschaft einer Figur und ist somit ein Attribut. Die Größe einer Figur ist zu wenig komplex, als daß sich eine Implementierung als Klasse lohnen würde. Sie kann ganz einfach durch zwei Zahlen ausgedrückt werden, die die Höhe und Breite der Figur angeben. Die Verben der Spezifikation sind *zeichnen*, *löschen* und *verändern*. Sie deuten auf Methoden der Klasse *Figur* hin. Adjektive gibt es in diesem Teil der Spezifikation nicht.

Schwachpunkte der Methode von Abbott

Wir sehen sofort, daß dieses Verfahren keinen vollständigen Klassenentwurf liefert, sondern höchstens als Ausgangspunkt dienen kann. Die Gründe sind klar: Zum einen kann das Ergebnis nur so gut sein wie die Spezifikation. Eine unvollständige Spezifikation enthält nicht alle Substantive, Verben und Adjektive und führt daher nicht zu den gewünschten Klassen. Zum anderen ist nicht jedes Substantiv eine Klasse und nicht jedes Verb eine Methode. Man muß die *relevanten* Wörter herausfiltern, was nicht immer einfach ist.

Anfänger machen oft den Fehler, *zu viele* Klassen zu bilden, also auch solche, die gar keine komplexen Daten enthalten oder für die es keine interessanten Zugriffsoperationen gibt.

10.3.4 CRC-Karten

CRC-Karten

Als Hilfsmittel für den Klassenentwurf werden in der Literatur oft sogenannte *CRC-Karten* (*Class-Responsibility-Collaboration*-Karten) empfohlen [BeC89]. Es handelt sich dabei um Karteikärtchen, auf denen die Aufgaben von Klassen und ihre Beziehungen zu anderen Klassen notiert werden (Abb. 10.4).

Klasse Zeichnung	
Aufgaben	**Partner**
Weiß, welche Figuren sie enthält. Zeichnet Figuren. Findet Figur an einer bestimmten Position. ...	Liste Figur

Abb. 10.4 CRC-Karte für eine Klasse „Zeichnung"

Für jede Klasse verwendet man eine Karte, die man mit dem Namen der Klasse beschriftet. In die linke Spalte trägt man die Aufgaben der Klasse ein, in die rechte die Partner-Klassen mit denen sie zusammenarbeitet. Die notierten Aufgaben müssen noch nicht unbedingt den Methoden entsprechen. Eine Aufgabe (z.B. verwalte Liste von Figuren) kann aus mehreren Methoden bestehen (z.B. *Insert*, *Delete*, *Broadcast*). Umgekehrt kann eine Methode auch mehrere Aufgaben lösen. Natürlich kann man, wenn man will, auch gleich die vollständige Klassenschnittstelle auf die Karte schreiben.

CRC-Karten haben verschiedene Vorteile: Sie sind einfach zu verstehen. Man kann mehrere Karten auf einem großen Tisch in verschiedenen Anordnungen ausbreiten und bekommt dadurch einen guten Überblick über die Aufgabenverteilung in einem Programm. Die beschränkte Größe der Karte zwingt dazu, die Klasse klein zu halten. Wenn man zwischen abstrakten Klassen und ihren konkreten Unterklassen unterscheiden will, legt man einen Kartenstapel an, mit der abstrakten Klasse zuoberst und den konkreten Klassen darunter.

10.4 Schnittstellenentwurf

Die Schnittstelle einer Klasse besteht aus denjenigen Attributen und Methoden, die für Klienten sichtbar sind. Die Schnittstelle einer Klasse *File* kann zum Beispiel folgendermaßen aussehen:

```
TYPE
   File = POINTER TO FileDesc;
   FileDesc = RECORD
      name-: ARRAY 64 OF CHAR;
      eof-, ok-: BOOLEAN;
      pos-: LONGINT;
      PROCEDURE (f: File) Open (name: ARRAY OF CHAR);
      PROCEDURE (f: File) Close;
      PROCEDURE (f: File) SetTo (pos: LONGINT);
      PROCEDURE (f: File) Read (VAR ch: CHAR);
      PROCEDURE (f: File) Write (ch: CHAR);
   END;
```

Ziel des Schnittstellenentwurfs

Ziel des Schnittstellenentwurfs ist es, den Wert einer Klasse zu maximieren und ihre Kosten zu minimieren. Unter *Kosten* versteht man die Implementierungs-, Änderungs-, Speicherplatz- und Laufzeitkosten. Der *Wert* einer Klasse wird bestimmt durch ihre Einfachheit, Allgemeinheit und Wiederverwendbarkeit. Er ist umso höher, je mehr man selbst und andere gewillt sind, die Klasse zu benutzen. Eine Schnittstelle sollte nach folgenden Gesichtspunkten entworfen werden [Hof90]:

Schnittstellenkriterien

1. *Konsistenz*. Halte dich konsequent an vorgegebene oder selbst aufgestellte Regeln. Die Regeln können die Parameterübergabe betreffen (z.B. Eingangsparameter vor Ausgangsparametern), die Namensgebung (z.B. konsistente Verwendung von Verben, Substantiven und Adjektiven) oder die Groß- und Kleinschreibung von Namen. Konsistenzregeln erleichtern es, den Rest eines Systems zu verstehen, wenn man bereits einen Teil davon kennt.

2. *Einfachheit.* Vermeide ausgefallene oder kompliziert zu benutzende Methoden. Je kleiner und natürlicher die Schnittstelle, desto handlicher die Klasse.

3. *Redundanzfreiheit.* Vermeide es, gleiche Dienste auf verschiedene Arten anzubieten: eliminiere redundante Methoden.

4. *Elementarität.* Fasse Operationen nicht zusammen, wenn sie auch einzeln benötigt werden.

5. *Wiederverwendbarkeit.* Schneide Klassen nicht auf bestimmte Klienten zu, sondern mache sie allgemein genug, so daß sie auch in anderem Kontext verwendbar sind.

6. *Robustheit gegenüber Änderungen.* Wähle die Schnittstelle einer Klasse so, daß sie unverändert bleibt, auch wenn sich ihre Implementierung ändert.

Einige Beispiele sollen diese Kriterien verdeutlichen.

Namensgebung

Namenskonventionen tragen zur besseren Lesbarkeit von Programmen bei. Sie werden in der Literatur selten ausdrücklich beschrieben, deshalb sind in Tabelle 10.1 einige Regeln angegeben, die sich bewährt haben.

Aus Gründen der Einheitlichkeit sollten Daten und Methoden, die ähnliche Dinge oder Operationen bezeichnen, gleich heißen. Die Operation, die ein Fenster, einen Rahmen oder eine Figur zeichnet, sollte in allen drei Fällen denselben Namen (z.B. *Draw*) tragen. Das erleichtert das Erlernen und Beherrschen neuer Klassen.

Redundanzfreiheit

Nehmen wir an, die Operation *text.Search(pattern, pos)* sucht ein Muster in einem Text ab der Position *pos*. Die Operation *text.SearchNext* sucht das nächste Auftreten des gleichen Musters ab

Tabelle 10.1 Bewährte Namenskonventionen

Namen für	beginnen mit	beginnen mit	Beispiele
Konstanten u. Variablen	Substantiv oder Adjektiv	Kleinbuchstaben Kleinbuchstaben	version, wordSize full
Typen	Substantiv	Großbuchstaben	File, TextFrame
Prozeduren	Verb	Großbuchstaben	WriteString
Funktionen	Substantiv oder Adjektiv	Großbuchstaben Großbuchstaben	Position Empty, Equal
Module	Substantiv	Großbuchstaben	Files, TextFrames

der Position, an der das Muster das letzte Mal gefunden wurde. *SearchNext* kann ohne weiteres durch *Search* ausgedrückt werden und sollte daher weggelassen werden.

Elementarität

Die Operation *file.Open(name, pos)* öffnet eine Datei und setzt die Leseposition auf *pos*. Diese Operation ist nicht elementar. Man bietet sie besser als zwei getrennte Operationen an, die man auch einzeln benutzen kann, also: *file.Open(name); file.SetTo(pos).*

Konflikte

Die genannten Kriterien können miteinander in Konflikt treten: Einerseits möchte man eine Klasse so allgemein wie möglich halten, um ihre Aussicht auf Wiederverwendung zu erhöhen; andererseits möchte man unnötige Methoden vermeiden. Wie soll man das unter einen Hut bringen? Oder: Man möchte nur elementare Operationen anbieten, die auf flexible Weise miteinander kombiniert werden können; andererseits muß man dann in Kauf nehmen, daß eine Aufgabe aus vielen Einzeloperationen besteht, die in der richtigen Reihenfolge aufgerufen werden müssen. In Fällen wie diesen muß man entscheiden, welches Kriterium einem wichtiger ist und muß das andere fallen lassen.

Virtuelle Sprachen

Guter Schnittstellenentwurf ist nicht leicht. Schließlich lassen sich mit Klassen *virtuelle Sprachen* schaffen, die neue Datentypen und neue Operationen enthalten. Schnittstellenentwurf ist daher nichts anderes als *Sprachentwurf*! Eine gute Sprache zu definieren ist schwer, daher ist es auch nicht verwunderlich, daß es schwer ist, gute Klassen zu entwerfen. Ob eine Klasse im Sinne der oben genannten Kriterien gut ist oder nicht, zeigt sich erst, sobald andere Personen als ihr Autor sie verwenden.

10.5 Abstrakte Klassen

Muster für Unterklassen

Abstrakte Klassen sind bereits aus Kapitel 6 bekannt. Sie enthalten Methoden ohne Implementierung, die in Unterklassen überschrieben werden müssen. Beim Bau erweiterbarer Software-Systeme kommt abstrakten Klassen große Bedeutung zu: sie sind der *Entwurf ihrer Unterklassen* – eine Schablone, die die Schnittstelle aller zukünftigen Unterklassen festlegt.

Bei der Implementierung von Benutzeroberflächen legt etwa eine abstrakte Klasse *GUIObject* fest, daß alle konkreten Unterklassen (*Button*, *CheckBox*, *ScrollBar*, etc.) die Meldungen *Draw*, *Move* und *Resize* verstehen müssen.

```
TYPE
   GUIObject = POINTER TO GUIObjectDesc;
   GUIObjectDesc = RECORD
      PROCEDURE (x: GUIObject) Draw;
```

```
    PROCEDURE (x: GUIObject) Move (dx, dy: INTEGER);
    PROCEDURE (x: GUIObject) Resize (dx, dy: INTEGER);
END;
```

Konkrete Unterklassen wie *Button* erben die Schnittstelle der abstrakten Klasse. Sie verstehen die gleichen Meldungen und können daher überall dort eingesetzt werden, wo ein *GUIObject* erwartet wird.

Abstrakte Klassen sind dazu da, andere Klassen von ihnen abzuleiten. Konkrete Klassen sind dazu da, Objekte von ihnen anzulegen. Abstrakte Klassen sind wiederverwendbares Design: man kann sie als Muster für neue *GUIObject*-Arten ansehen. Konkrete Klassen sind hingegen oft auf einen bestimmten Zweck zugeschnitten und daher nicht so leicht für andere Zwecke verwendbar; man kann auch seltener neue Klassen aus ihnen ableiten. Je mehr abstrakte Klassen man also findet, desto mehr wiederverwendbare Abstraktionen erhält man.

Wie findet man abstrakte Klassen?

Wie kommt man zu abstrakten Klassen? Entweder man sieht von vornherein, daß es von einer Klasse verschiedene Varianten gibt und schält deren gemeinsames Verhalten heraus. Das ist bei generischen Bausteinen (Kapitel 8.2), bei heterogenen Datenstrukturen (Kapitel 8.3) und bei austauschbarem Verhalten (Kapitel 8.4) der Fall. Man kann auch von einer konkreten Klasse ausgehen, die sich bewährt hat, und versuchen, daraus eine wiederverwendbare Abstraktion zu machen. Angenommen, man hat eine Klasse *BarChart* für Balkendiagramme. Die wiederverwendbare Abstraktion ist hier aber nicht das Balkendiagramm, sondern ein *allgemeines Diagramm*. Man kann daher die charakteristischen Eigenschaften von Diagrammen herausziehen und in einer abstrakten Klasse *Chart* zusammenfassen, von der *BarChart* ein Spezialfall ist. Man bedenke: es geht vor allem darum, die *Schnittstelle* wiederverwendbar zu machen und nicht den *Code*.

Wie macht man abstrakte Klassen wiederverwendbar?

Um Klassen wirklich wiederverwendbar zu machen, darf man sich nicht mit ihrer ersten Fassung zufriedengeben. Man muß sie immer wieder überarbeiten. Nur so kann man ihren Wert erhöhen. *Johnson* schreibt dazu sinngemäß [JoF88]: Wiederverwendbarkeit entsteht nicht von selbst. Genauso wichtig, wie eine neue Klasse zu entwerfen, ist es, eine bestehende Klasse zu überarbeiten, um sie einfacher und wiederverwendbarer zu gestalten. Erfahrene Programmierer wenden genausoviel Zeit dafür auf, *alte* Klassen zu vereinfachen, wie *neue* Klasen zu schreiben. Nützliche Abstraktionen werden meist von Leuten gefunden, die eine Vorliebe für einfache Dinge haben, die gewillt sind, Klassen mehrmals umzuschreiben, um verständlichen und wiederverwendbaren Code zu erhalten.

Ob eine Klasse wiederverwendbar ist, zeigt sich erst, nachdem sie tatsächlich wiederverwendet wurde. Eine Klasse, die nicht mehrmals und von unterschiedlichen Leuten wiederverwendet wurde, ist nicht wiederverwendbar.

10.6 Wann sind Klassen sinnvoll?

In Sprachen wie *Smalltalk* gibt es keine anderen Datentypen als Klassen und keine anderen Operationen als Methoden. In hybriden Sprachen wie *Oberon-2* sind Klassen jedoch nur *einer* von vielen Bausteinen. Man hat daneben noch einfache Datentypen (INTEGER, CHAR), zusammengesetzte Datentypen (Arrays, Records), abstrakte Datentypen und Module. Oft ist ein Array einfacher und natürlicher als eine Klasse, eine Prozedur besser als eine Methode.

Es stellt sich also die Frage: Wann sind Klassen sinnvoll und wann nicht? Klassen sind nach der Meinung des Verfassers nur dann gerechtfertigt, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

1. *Wenn die Daten genügend komplex sind, so daß sich eine Kapselung lohnt.* Klassen sollen komplexe Daten vereinfachen, indem sie Details verbergen. Die abstrakte Sicht, die eine Klasse anbietet, muß wesentlich einfacher sein als die konkrete Datenstruktur, die sie kapselt. Eine Klasse *Speed* hat zum Beispiel wenig Sinn, denn eine Geschwindigkeit läßt sich viel einfacher durch eine gewöhnliche Zahl ausdrücken. Eine Klasse *File* hingegen ist nützlich, denn sie verbirgt für Klienten unwichtige Details, wie einen Datenpuffer, eine Position, Zugriffsrechte usw. Die Benutzung der Abstraktion *File* ist einfacher als die Benutzung ihrer konkreten Datenstruktur.

   Daten genügend komplex

2. *Wenn es genügend sinnvolle Operationen mit den Daten gibt.* Wenn einem als Operationen nur das Setzen und Abfragen von Attributen einfallen, dann ist meist ein Record das geeignetere Konstrukt. Für eine Klasse *Speed* gibt es keine interessanten Operationen. Man kann einen Wert setzen und abfragen und vielleicht noch Geschwindigkeiten addieren. Das kann man aber mit Zahlen auch. Eine Methode *Add* ist nicht einfacher zu verstehen als die Standardoperation +, eher im Gegenteil. Eine Klasse *File* hingegen besitzt viele sinnvolle Operationen: öffnen, schließen, lesen, schreiben, usw. Klassen mit nur einer einzigen Methode sind verdächtig. Sie können zwar in Ausnahmefällen sinnvoll sein, nämlich dann, wenn sie außer der Methode noch einen Zustand haben (z.B. ein Zufallszahlengenerator). Meist ist aber eine Prozedur das bessere Konstrukt.

   Sinnvolle Operationen

3. *Wenn die Daten in Varianten existieren.* Viele der nützlichsten Anwendungen der objektorientierten Programmierung beruhen auf

   Varianten

Situationen, in denen mit Varianten einer Basisklasse gearbeitet wird. Wenn man in seinem Programm Daten ausmachen kann, die in Varianten existieren und vom Programm nicht unterschieden werden sollen, dann sind das typische Kandidaten für Klassen. Man kann so die dynamische Bindung ausnutzen und hält sich die Möglichkeit offen, später neue Varianten hinzuzufügen, ohne die Algorithmen ändern zu müssen, die schon mit den bisherigen Varianten arbeiten. Die Gleichbehandlung von Varianten ist vielleicht sogar das wichtigste Motiv für den Einsatz von Klassen, weil es anders kaum möglich ist, ein Programm in die Lage zu versetzen, mit *neuen* Varianten zu arbeiten, ohne es zu ändern oder zumindest neu übersetzen zu müssen.

Aussicht auf Wiederverwendung

4. *Wenn Aussicht auf Erweiterung und Wiederverwendung besteht.* Manche Bausteine sind so allgemein, daß sie nicht nur in demjenigen Programm verwendet werden können, in dem sie entstanden sind, sondern auch in anderen Programmen. Ein Beispiel sind Popup-Menüs. Sie sind anwendungsunabhängig, wiederverwendbar und erweiterbar (z.B. mehrstufige Menüs).

In den meisten anderen als den oben genannten Fällen sind Klassen nicht sinnvoll:

Wenn die Daten einfach sind, reichen Arrays, Records oder Sets aus. Sie sind genauso leicht zu verstehen wie Klassen (oder sogar einfacher) und sind überdies effizienter. Ein Array, von dem Anzahl und Typ seiner Elemente bekannt sind, sollte nicht als Klasse implementiert werden.

Wenn Daten anwendungsspezifisch sind und nur lokal in einer Prozedur verwendet werden, lohnen sich Klassen dafür meist nicht. Beim Formatieren eines Textes braucht man zum Beispiel eine Zwischendatenstruktur, um die Wortlängen und Wortzwischenräume zu speichern. Arrays und Records sind dafür meist ausreichend. Aus Gründen der Datenabstraktion kann freilich auch für anwendungsspezifische Daten eine Klasse gerechtfertigt sein.

Durch Datenabstraktion tragen Klassen dazu bei, die Komplexität eines Programms zu verringern. Man sollte allerdings bedenken, daß jede Klasse auch ein gewisses Maß an neuer Komplexität *einführt*. Man muß die Schnittstelle und Bedeutung ihrer Operationen verstehen; die Implementierung einer Klasse kostet außerdem Code, der ein Programm vergrößert und damit die Fehlerwahrscheinlichkeit erhöht.

Klassen sind nur *ein* Konstrukt neben vielen anderen. Sie erlauben in vielen Fällen elegante Lösungen. Das ist aber noch kein Grund, *alles* mit Klassen auszudrücken. Dies ist ähnlich wie bei Rekursion: Auch Rekursion ist eine Technik, mit der man manche Algorithmen

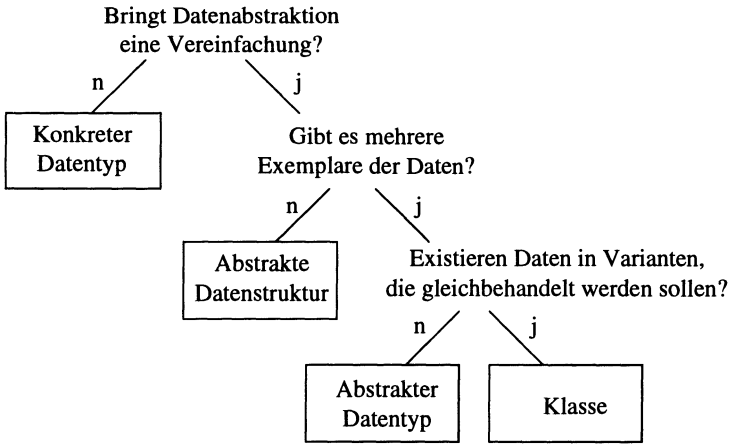


Abb. 10.5 Auswahl des geeigneten Modularisierungskonstrukts

elegant formulieren kann. Das heißt aber nicht, daß man *alle* iterativen Abläufe rekursiv implementieren sollte. Meist ist eine Schleife viel natürlicher und effizienter.

Das Oberon-System selbst besteht nur zum Teil aus Klassen. Der größere Teil besteht aus abstrakten Datenstrukturen, abstrakten Datentypen und gewöhnlichen Prozeduren. Trotzdem ist es modular und erweiterbar. Abb. 10.5 zeigt, wie das geeignete Konstrukt zur Datenmodellierung gewählt werden soll.

Klassen nicht um jeden Preis

Das Fazit ist also: Klassen sollen nicht um jeden Preis eingesetzt werden, sondern nur dann, wenn sie ein Programm übersichtlicher und erweiterbar machen und wenn diese Erweiterbarkeit notwendig ist. Flexibilität hat ihren Preis, und wer ist schon bereit, einen Preis für etwas zu bezahlen, das er nicht ausnutzt? Es ist die Kunst eines erfahrenen Programmierers, zu erkennen, wann Klassen sinnvoll sind und wann nicht.

10.7 Häufige Entwurfsfehler

Guten Entwurf zu lehren ist schwierig, wenn nicht sogar unmöglich. Manchmal ist es leichter zu zeigen, wie man Programme *nicht* entwerfen sollte. Auch diese Information ist für den Leser nützlich. Indem man die gröbsten Fehler vermeidet, kommt man schon zu passablen Entwürfen. Dieses Kapitel beschreibt daher einige der häufigsten Entwurfsfehler:

- Zu viele triviale Klassen
- Verwechslung von Ist-Beziehung und Benutzt-Beziehung
- Verwechslung von Oberklasse und Unterklasse
- Varianten mit gleicher Struktur und gleichem Verhalten
- Falsches Empfängerobjekt
- Zu tiefe oder zu flache Klassenhierarchie

Fehler wie diese unterlaufen vor allem Anfängern – aber nicht nur ihnen. Man findet sie sogar in manchen Büchern über objektorientierte Programmierung.

10.7.1 Zu viele triviale Klassen

Wir wissen bereits aus dem letzten Kapitel, daß Klassen nicht um jeden Preis eingesetzt werden sollten. Es ist ein häufiger Anfängerfehler, für jeden noch so einfachen Begriff eine Klasse zu wählen. Klassen wie *Salary* oder *Amount* blähen ein Programm nur auf, ohne seine Komplexität zu verringern oder wesentliche Flexibilität zu bieten. Einfache Integer-Zahlen reichen dafür vollkommen aus.

In diesen Fällen ist es klar, daß Klassen nicht das richtige Konstrukt sind. In anderen Fällen ist es weniger offensichtlich, wie zum Beispiel bei einer Uhrzeit. Soll man dafür ein gewöhnliches Record wählen:

```
TYPE
   Time = RECORD
      hours: INTEGER;
      minutes: INTEGER;
      seconds: INTEGER
   END
```

oder besser eine Klasse:

```
TYPE
   Time = RECORD
      PROCEDURE (VAR t: Time) Get (VAR h, m, s: INTEGER);
      PROCEDURE (VAR t: Time) Set (h, m, s: INTEGER);
      PROCEDURE (VAR t: Time) Add (t1: Time);
      PROCEDURE (VAR t: Time) Subtract (t1: Time);
   END
```

Das kommt darauf an, was man mit Uhrzeiten machen will. Wenn man sie nur lokal in einem Programm verwendet und nicht mit ihnen rechnet, genügt ein Record. Es ist genügend einfach zu verstehen und effizient im Zugriff. Wenn man die Uhrzeit jedoch als einen wieder-

verwendbaren Baustein sieht, der auch in anderen Programmen eingesetzt werden soll, und wenn man Zeiten addieren und subtrahieren will, dann ist ein abstrakter Datentyp oder eine Klasse das Richtige. Ein Baustein dieser Art läßt außerdem die Möglichkeit offen, Uhrzeiten später einmal in anderer Form zu speichern, ohne daß Klienten von der Änderung betroffen sind. Es kommt also darauf an, was man mit den Daten vorhat.

10.7.2 Verwechslung von Ist- und Benutzt-Beziehung

Vererbung stellt eine Ist-Beziehung zwischen Unterklasse und Oberklasse dar. *B* darf nur dann von *A* abgeleitet werden, wenn es eine Erweiterung oder Verfeinerung von *A* ist. Die Vererbung wird jedoch oft mißbraucht, um eine Benutzt- oder Hat-Beziehung darzustellen. Abb. 10.6 zeigt ein Beispiel.

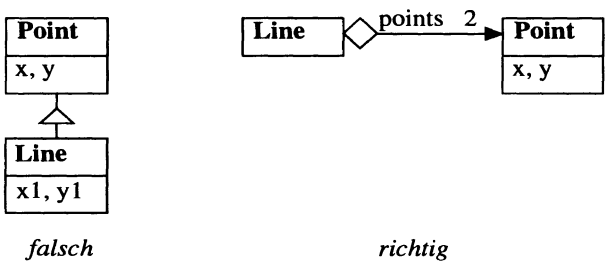


Abb. 10.6 Falsche Ist-Beziehung

Die Idee des Diagramms auf der linken Seite ist offenbar, daß eine Linie durch ihre beiden Endpunkte beschrieben werden kann. Die Koordinaten des einen Punkts erbt man, die des anderen fügt man in *Line* hinzu. Diese Sichtweise ist aber falsch! Eine Linie *ist* kein Punkt. Sie *hat* zwei Punkte. Es muß also wie in Abb. 10.6 rechts aussehen.

Dieser Fehler tritt manchmal auch in subtilerer Form auf (Abb. 10.7 links). Hier trifft es zwar zu, daß ein Fenster in der Regel ein Rechteck ist, trotzdem schränkt die Ist-Beziehung die Flexibilität von *Window* ein. Vielleicht möchte man später einmal ovale Fenster haben. Dann *ist* ein Fenster kein Rechteck mehr, sondern *benutzt* vielmehr eine bestimmte Form, die ein Rechteck oder ein Oval sein kann (Abb. 10.7 rechts).

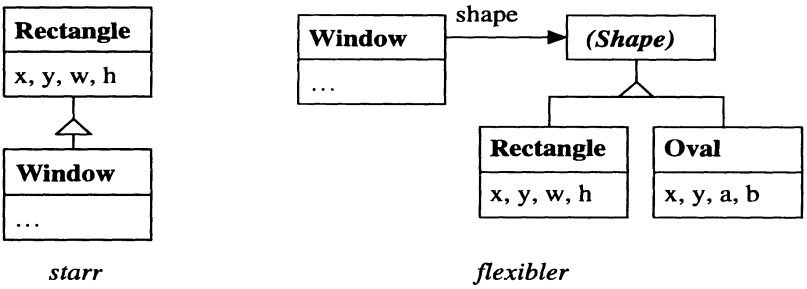


Abb. 10.7 Unflexible Ist-Beziehung

Wenn man *Window* aus *Rectangle* ableitet, hat das außerdem den Nachteil, daß man es nicht mehr von einer anderen Klasse ableiten kann, ohne mehrfache Vererbung zu benutzen. Zum Beispiel könnte es nötig sein, *Window* auch aus einer Klasse *ListNode* abzuleiten, damit man verschiedene Fenster in einer Liste zusammenhängen kann.

10.7.3 Verwechslung von Oberklasse und Unterklasse

Manchmal ist es gar nicht leicht zu sagen, welche von zwei Klassen die Oberklasse und welche die Unterklasse sein soll. Abb. 10.8 gibt dafür ein Beispiel.

Man kann argumentieren, daß ein Rechteck eine Erweiterung eines Quadrats ist, denn während zur Speicherung eines Quadrats ein Eckpunkt und eine Seitenlänge nötig sind, braucht man zur Speicherung eines Rechtecks dieselben Daten und eine zweite Seitenlänge.

Dieses Argument ist aber falsch, denn nicht jedes Rechteck ist ein Quadrat. Das Gegenteil ist richtig: jedes Quadrat ist ein Rechteck! Die Unterklasse ist eine Spezialisierung der Oberklasse. Man muß die Klassenbeziehung immer so wählen, daß sich eine Ist-Beziehung ergibt. Nur dann kann man Objekte der Unterklasse überall dort verwenden, wo ein Objekt der Oberklasse erwartet wird.

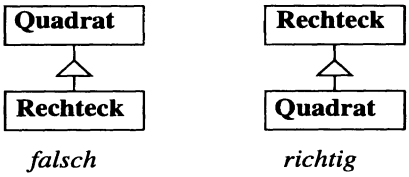


Abb. 10.8 Ist *Rechteck* eine Unterklasse von *Quadrat* oder umgekehrt?

10.7.4 Varianten mit gleicher Struktur und gleichem Verhalten

Manche Programmierer neigen dazu, verschiedene Objektmengen einer Klasse, die zwar gleiche Struktur und gleiches Verhalten aufweisen, aber sich durch den Wert eines Attributs voneinander unterscheiden, als separate Unterklassen anzusehen. Die Klassenhierarchie in Abb. 10.9 (links) ist im allgemeinen falsch:

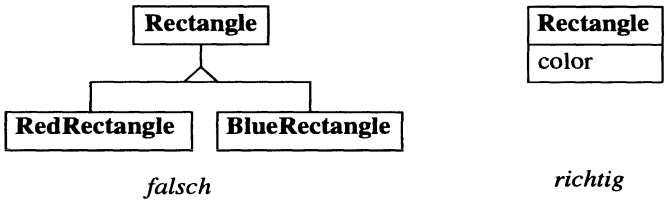


Abb. 10.9 Unterklasse mit gleicher Struktur und gleichem Verhalten

Wenn rote und blaue Rechtecke die gleiche Art von Attributen und die gleichen Methoden besitzen, gehören sie beide zur Klasse *Rectangle*. Sie unterscheiden sich nur durch den Wert eines Attributs, das ihre Farbe angibt (Abb. 10.9 rechts). Eine Unterscheidung in zwei Unterklassen wäre hingegen dann angebracht, wenn rote und blaue Rechtekke verschiedene Attribute hätten oder auf eine Meldung unterschiedlich reagieren würden.

10.7.5 Falsches Empfängerobjekt

Manchmal ist man im Zweifel, welcher Klasse man eine Operation zuordnen soll. Um zum Beispiel Elemente aus einer Liste zu entfernen, braucht man eine Methode *Remove*. Gehört diese Methode zu Listen oder zu Elementen? Heißt es also

```
list.Remove(element)
```

oder

```
element.RemoveFrom(list)
```

Man kann argumentieren, daß Elemente autonom sein sollen und daher wissen müssen, wie sie sich aus einer Liste zu entfernen haben. Diese Sichtweise ist aber falsch. Das Entfernen von Objekten ist eine Listenoperation.

Der Empfänger einer Meldung muß immer dasjenige Objekt sein, das durch die Operation verändert wird. In diesem Beispiel wird die Liste verändert und nicht das Element. Der Zustand einer Liste darf nur durch ihre eigenen Methoden verändert werden, sonst verletzt man das Geheimnisprinzip und kann keine Invarianten über den Zustand der Liste mehr garantieren.

Was ist aber, wenn eine Methode die Daten von Objekten mehrerer Klassen verändert? Welcher Klasse soll man sie dann zuordnen? Eine Situation dieser Art deutet meist auf einen Entwurfsfehler hin. Die Methode sollte in mehrere Methoden aufgespalten werden, die jeweils nur die Daten ihres Empfängerobjekts verändern.

10.7.6 Zu tiefe oder zu flache Klassenhierarchie

Man kann zwar schwer sagen, wie tief eine Klassenhierarchie sein soll, aber zu tiefe oder zu flache Klassenhierarchien sind im allgemeinen unerwünscht.

Zu tiefe Hierarchien treten oft auf, wenn man nicht nur abstrakte sondern auch konkrete Klassen erweitert und vor allem auf Wiederverwendung von Code aus ist. Das Problem bei zu tiefen Hierarchien ist, daß jede Methode kaum noch selbst etwas leistet, bevor sie die gleichnamige Methode der Oberklasse aufruft. Eine Operation ist so auf viele Methoden verteilt, was die Wartung und Fehlersuche erschweren kann.

Zu flache Hierarchien treten auf, wenn abgeleitete Klassen wenig oder nichts von ihrer Basisklasse wiederverwenden. Im Extremfall gibt es nur eine einzige abstrakte Klasse Object, aus der alle anderen Klassen abgeleitet sind. Das ist sicher falsch, denn man verliert dadurch fast alle Vorteile der Objektorientiertheit.

Eine Klassenhierarchie sollte ausgewogen sein. Die inneren Knoten sollen für abstrakte Klassen stehen, die Blätter für konkrete. Aus einer abstrakten Klasse werden häufig viele konkrete Klassen abgeleitet; hier geht der Baum in die Breite. Hingegen ist der Verzweigungsgrad beim Ableiten abstrakter Klassen aus anderen abstrakten Klassen gering; hier geht der Baum in die Tiefe (Abb. 10.10).

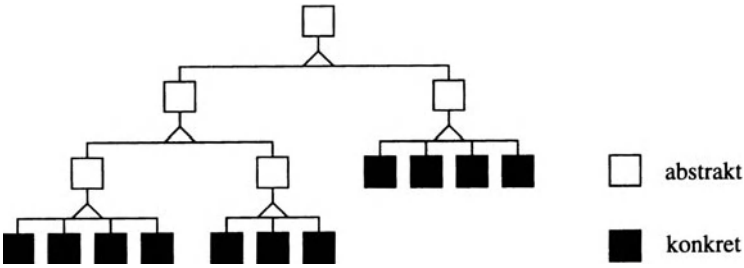


Abb. 10.10 Klassenhierarchien sollten weder zu tief noch zu flach sein

11 Frameworks

Objektorientierte Systeme werden meist nicht von Grund auf neu geschrieben, sondern bauen auf bestehenden Systemen auf. Objektorientiertes Programmieren bedeutet also meist Erweitern eines bestehenden Systems. In der Erweiterung liegt auch der größte Nutzen dieser Technik. Erst wenn Programme auf Vorhandenes aufbauen können, steigt die Produktivität des Programmierers.

Schon wenn man eine *einzelne* Klasse aus einer bestehenden ableiten kann, spart man viel Arbeit. Wenn man jedoch ein *ganzes System* von Klassen wiederverwenden kann, ist der Nutzen noch viel größer.

Ein erweiterbares System zusammenspielender Klassen nennt man im Englischen ein *Framework* [Deu89], [Pre96]. Im Deutschen gibt es keine gute Übersetzung dafür. Machmal hört man den Begriff *Gerüst* oder *Programmrahmen*. Der Begriff Framework ist aber verbreiteter, weshalb wir ihn in der Folge verwenden.

Wir wollen uns in diesem Kapitel mit der Idee von Frameworks vertraut machen und uns dann einige Beispiele ansehen.

11.1 Frameworks als erweiterbare Systeme

Ein Framework ist ein objektorientiertes *Software-Halbfabrikat*, das zu verschiedenen Endfabrikaten ausgebaut werden kann. Es besteht aus kooperierenden Objekten, die eine bestimmte Grundaufgabe erfüllen. An einige Stellen weist es "*Steckplätze*" auf, an denen der Programmierer eigene Objekte einstecken und somit die Funktionalität des Frameworks seinen Bedürfnissen anpassen kann (Abb. 11.1).

Hot Spots

Die Steckplätze nennt man "*Hot Spots*" [Pre96]. Sie sind die interessanten Punkte im Framework, die Knöpfe, an denen man drehen kann, um das Framework anzupassen. Das eigentliche Framework stellt hingegen einen "*Frozen Spot*" dar, der nicht verändert werden kann. Hier steckt die Steuerlogik des Frameworks, die dafür sorgt, daß gewissen Grundaufgaben erfüllt und bei Bedarf die Erweiterungen in den Steckplätzen aktiviert werden.

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

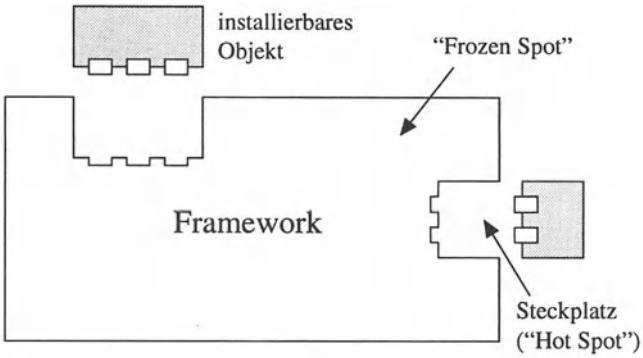


Abb. 11.1 Framework: Software-System mit Steckplätzen

Jeder Steckplatz ist durch eine *abstrakte Klasse* im Framework definiert. Sie bestimmt die „Form" der Objekte, die dort eingesteckt werden können (d.h. ihren Typ). Der Programmierer leitet aus der abstrakten Klasse konkrete Unterklassen ab, deren Objekte dann mit dem Steckplatz kompatibel sind.

Wie kommt man zu Frameworks

Wie kommt man zu Frameworks? Im Prinzip kann jede Benutzt-Beziehung zwischen zwei Klassen zu einem Mini-Framework ausgebaut werden. Abb. 11.2 zeigt zum Beispiel eine Klasse *A*, die eine Klasse *B* benutzt, indem sie die Methode *Q* aufruft.

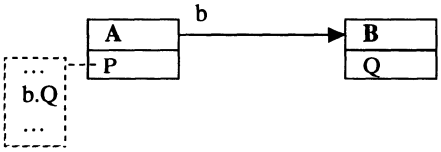


Abb. 11.2 Starre Benutzt-Beziehung

Diese Beziehung ist starr. Ein *A*-Objekt benutzt immer *genau ein B*-Objekt. Will man die Beziehung flexibler gestalten und statt *B* verschiedene *Varianten* von *B* zulassen, muß man *B* zu einer abstrakten Klasse und *Q* zu einer abstrakten Methode machen (Abb. 11.3).

A-Objekte können nun sowohl mit *B1*- als auch mit *B2*-Objekten arbeiten. Der Steckplatz ist die abstrakte Klasse *B*, in die zum Beispiel ein *B1*-Objekt eingesteckt werden kann, indem man es einer Framework-Variablen vom Typ *B* zuweist:

```
...
NEW(b1);      (* create new B1 object *)
a.b := b1;    (* plug it into the hot spot of the framework *)
...
```

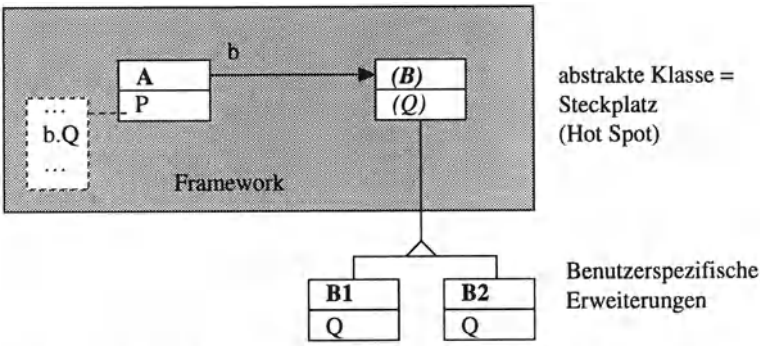


Abb. 11.3 Flexible Benutzt-Beziehung in einem Framework

Das Framework läßt die konkreten Erweiterungen (*B1* und *B2*) offen und spezifiziert nur ihre *Schnittstelle*. Die abstrakte Klasse *B* kann allerdings bereits gewisses Verhalten implementieren, das allen *B*-Varianten gemeinsam ist. Auch die Klasse *A* implementiert bereits eine gewisse Steuerlogik, die dafür sorgt, daß die *Q*-Methode von *B* zum richtigen Zeitpunkt aufgerufen wird. All das wird wiederverwendet, wenn man das Framework benutzt und anpaßt.

Ein Framework ist nützlicher als eine Bibliothek von Prozeduren oder Modulen. Prozedurbibliotheken bieten lediglich einzelne Operationen an, aber keine Hinweise, wie diese Operationen zu einem sinnvollen System zusammengesetzt werden müssen. Die Toolbox des Apple Macintosh ist ein Beispiel dafür. Jeder der diese Bibliothek schon einmal benutzt hat, weiß, wie schwierig es ist, die für eine Aufgabe benötigten Prozeduren zu finden und in der richtigen Reihenfolge aufzurufen.

Frameworks versus Prozedurbibliotheken

Frameworks stellen die Architektur herkömmlicher Programme auf den Kopf (Abb. 11.4). In herkömmlichen Programmen schreibt der Programmierer ein Hauptprogramm, das Prozeduren aus einer Bibliothek aufruft. Bei der Verwendung von Frameworks ist es umgekehrt: Hier stammt das eigentliche Hauptprogramm mit der Steuerlogik (das Framework) aus einer Bibliothek und ruft zu gegebener Zeit Methoden

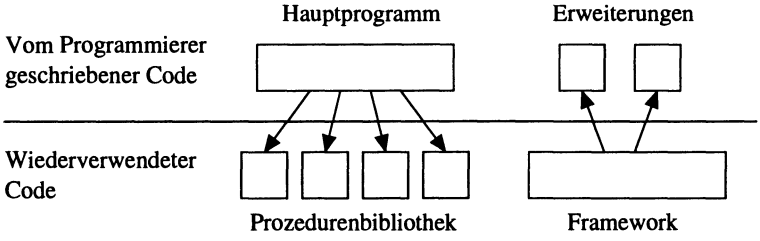


Abb. 11.4 Architektur von Prozedurbibliotheken und Frameworks

auf, die der Programmierer zur Verfügung stellt. Es ist nicht das Anwendungsprogramm, das Bibliotheksroutinen ruft, sondern Bibliotheksroutinen rufen Teile des Anwendungsprogramms auf. Man bezeichnet diese Situation als das Hollywood-Prinzip: "dont call us, we'll call you".

Frameworks als Entwurf von Systemen

Frameworks sind immer auf einen bestimmten Anwendungsbereich zugeschnitten. Es gibt zum Beispiel Frameworks für grafische Benutzeroberflächen, für Simulationsaufgaben oder für Betriebssysteme. Der Entwurf eines Frameworks erfordert Erfahrung und Fachwissen im jeweiligen Anwendungsbereich. Nur so ist man in der Lage, die Gemeinsamkeiten mehrerer Anwendungen in diesem Bereich herauszuziehen und in einem Framework zu implementieren.

Während eine *abstrakte Klasse* der Entwurf ihrer konkreten Unterklassen ist, ist ein *Framework* der Entwurf aller Teilsysteme, zu denen es erweitert werden kann. So, wie eine abstrakte Klasse die Verallgemeinerung einer konkreten Klasse ist, ist ein Framework die Verallgemeinerung eines Systems von Klassen.

11.2 Ein Framework für die Menüauswahl

Betrachten wir nun ein Beispiel eines Frameworks. Interaktive Programme benutzen üblicherweise *Popup-Menüs* (oder *Pulldown-Menüs*) zur Auswahl von Aktionen. Wenn der Benutzer mit der Maus in den Menübalken klickt, müssen folgende Aktionen ablaufen:

1. Menü aufklappen und anzeigen.
2. Maus verfolgen und die getroffenen Menüeinträge invertieren.
3. Wenn der Mausknopf losgelassen wird, ausgewählten Menüeintrag bestimmen.
4. Menüeintrag behandeln.

Von all diesen Aktionen ist nur die vierte anwendungsabhängig. Die ersten drei laufen bei allen Menüs gleich ab. Man kann daher die vier Aktionen in einer *Schablonenmethode* implementieren (siehe Kapitel 9.4.4), in der die Behandlung des Menüeintrags (d.h. Aktion 4) einer abstrakten Methode überlassen wird.

Bevor wir das tun, überlegen wir uns aber noch, welche Klassen wir für Menüs benötigen. Ein Menü besteht aus einer Folge von Menüeinträgen, wie das in Abb. 11.5 dargestellt ist. Um möglichst flexibel zu bleiben, wollen wir als Menüeinträge nicht nur Text, sondern auch Grafiken zulassen, damit wir zum Beispiel die Auswahl eines Farbmusters implementieren können.

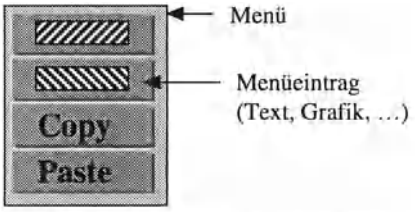


Abb. 11.5 Bestandteile eines Menüs

Diese Überlegungen führen uns zu einer Klasse *Menu* für das gesamte Menü und einer abstrakten Klasse *MenuItem*, von der konkrete Menüeinträge für Text und Grafik abgeleitet werden können. Das Framework und seine Steuerlogik sind in Abb. 11.6 dargestellt.

Wenn der Benutzer auf das Menü im Menübalken klickt, wird die Methode *Select* der Klasse *Menu* aufgerufen. Sie ist eine Schablonenmethode, die das Menü aufklappt (*self.Show*), die Mausbewegungen verfolgt (*self.TrackMouse*), das Menü wieder zuklappt (*self.Hide*) und schließlich den ausgewählten Menüeintrag behandelt (*item.Handle*).

MenuItem ist eine abstrakte Klasse, von der es zwei Unterklassen *TextItem* und *GraphicItem* gibt. Diese beiden Klassen sind ebenfalls (teilweise) abstrakt. *TextItem* speichert zwar bereits den Text des Menüeintrags (z.B. "Copy") und kann diesen Text in der Methode *Show* anzeigen (dieses Verhalten ist allen Texteinträgen gemeinsam),

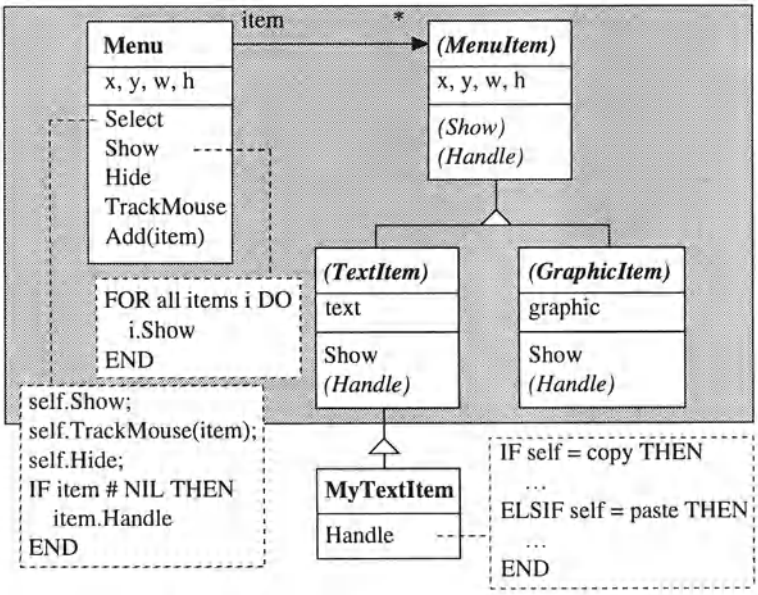


Abb. 11.6 Framework zur Menübehandlung

sie kann aber noch keine Menüeinträge behandeln. Dazu muß der Programmierer erst eine Unterklasse *MyTextItem* ableiten und dort die Methode *Handle* implementieren. Diese Methode prüft dann anhand einer Fallunterscheidung, welcher Eintrag getroffen wurde und reagiert entsprechend.

Das Framework besteht aus den Klassen im grauen Kästchen. Das einzige, was der Programmierer ergänzen muß, ist die Klasse *MyTextItem*, die außerhalb des Frameworks liegt. Um das Framework zu initialisieren, wird er etwa folgendes tun:

```
VAR
   copy, paste: MyTextItem;
...
NEW(copy);
copy.text := "Copy";
menu.Add(copy);
NEW(paste);
paste.text := "Paste";
menu.Add(paste);
```

Dadurch sind die beiden Einträge im Menü installiert. Alles andere geschieht nun automatisch. Wenn der Benutzer mit der Maus den Eintrag "Copy" auswählt, wird die *Handle*-Methode des *copy*-Eintrags aufgerufen, die entsprechend reagiert.

Eine andere Implementierungsmöglichkeit besteht darin, *Handle* nicht als Methode, sondern als *Prozedurvariable* zu implementieren. Dann könnte man in jedem Eintrag eine eigene Prozedur installieren (eine für Copy, eine für Paste, etc.) und bräuchte keine Fallunterscheidung mehr durchzuführen.

Dieses Beispiel zeigt, daß man mit Frameworks relativ komplexe Aufgaben mit wenig Implementierungsaufwand lösen kann. Der Programmierer muß nur die Klasse *MyTextItem* implementieren. Alles andere – einschließlich der Steuerlogik der Menüverwaltung – kann wiederverwendet werden. Das ist ein enormer Produktivitätsgewinn.

Allerdings erkennt man an diesem Beispiel auch bereits das Hauptproblem von Frameworks, nämlich daß man sie kaum ohne gute Dokumentation benutzen kann. Der Programmierer muß genau wissen, welche Klassen er erweitern und welche Methoden er überschreiben muß. Er braucht Informationen über den Zustand, der gilt, wenn seine Methoden vom Framework gerufen werden. All das erfordert Einarbeitungsaufwand. Ist man aber bereit, diese Zeit zu investieren, sind Frameworks eine nützliche Sache.

11.3 Das MVC-Schema

Eines der am häufigsten benutzten Frameworks in interaktiven Programmen ist das *Model-View-Controller-Framework* (*MVC-Framework*) [KrP88]. Eigentlich ist es mehr ein *Muster* im Sinne der Entwurfsmuster aus Kapitel 9 als ein Framework aus Code. Obwohl es in manchen Bibliotheken als Code vorliegt, wird es oft auch völlig neu implementiert, wobei nur die Architektur dieses Musters wiederverwendet wird.

Bestandteile

Worum geht es? Interaktive Programme bestehen grob betrachtet meist aus drei Teilen:

1. *Modell* (*model*): der Programmteil, der die bearbeiteten Daten verwaltet, z.B. Text, Grafik, Tabellen, etc.
2. *Sicht* (*view*): der Programmteil, der für die Anzeige der Daten am Bildschirm verantwortlich ist.
3. *Eingabeteil* (*controller*): der Programmteil, der Benutzereingaben, wie Mausklicks oder Tastendrucke interpretiert.

Abb. 11.7 zeigt das Zusammenspiel dieser drei Teile (die Nummern werden später erklärt).

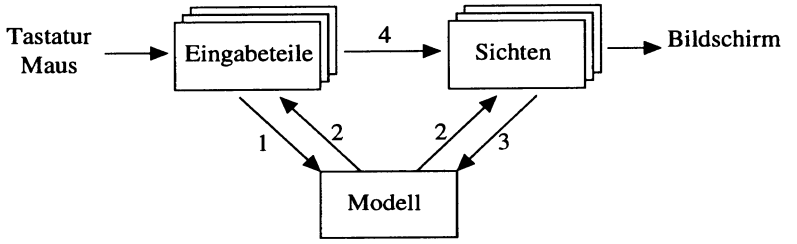


Abb. 11.7 Komponenten des MVC-Schemas

Zu einem Modell kann es mehrere Sichten geben: Zum Beispiel kann ein Texteditor mehrere Fenster haben, in denen er verschiedene Ausschnitte desselben Textes anzeigt. Zu einer Zahlenfolge kann es eine Sicht geben, die sie als Text und eine andere, die sie als Balkendiagramm darstellt. Alle Sichten zeigen dasselbe Modell. Wenn sich das Modell ändert, müssen daher alle Sichten nachgeführt werden. Der Hauptzweck des MVC-Schemas besteht darin, mehrere Sichten *konsistent nachzuführen*.

Sicht und Eingabeteil treten immer *paarweise* auf. Zu jeder Sicht gehört ein eigener Eingabeteil, da dieselbe Eingabe in verschiedenen Sichten unterschiedliche Auswirkungen haben kann. Ein Mausklick in

die Textsicht einer Zahlenfolge kann zum Beispiel bewirken, daß ein Textstück selektiert wird, während derselbe Klick in die Diagrammsicht bewirken kann, daß eine Grafik verschoben wird.

Zwischen Modell, Sichten und Eingabeteilen fließen folgende Meldungen (die Nummern entsprechen denen in Abb. 11.7):

Meldungen im MVC-Schema

1. Ein Eingabeteil reagiert auf Tastatureingaben oder Mausklicks, indem er das Modell verändert (z.B. indem es Zeichen in einen Text einfügt).
2. Das Modell teilt allen seinen Sichten mit, daß es verändert wurde und daß daher seine Darstellung nachgeführt werden muß. Auch die Eingabeteile werden von der Änderung benachrichtigt, denn es ist denkbar, daß eine Änderung des Modells dazu führt, daß Eingaben nun anders behandelt werden müssen.
3. Nachdem eine Sicht aufgefordert wurde, die Darstellung des Modells nachzuführen (z.B. ein eingefügtes Textstück zu zeichnen), holt sie sich die dazu benötigten Daten aus dem Modell und stellt sie auf dem Bildschirm dar.
4. In einigen Fällen spricht ein Eingabeteil die Sicht direkt an, zum Beispiel wenn ihr Inhalt verschoben werden soll (scrolling). Hier wird das Modell nicht verändert, sondern nur die Sicht verschoben.

Ein Eingabeteil darf eine Sicht niemals selbst verändern, sondern muß das immer über den Umweg des Modells tun. Nur so ist sichergestellt, daß andere Sichten dieses Modells ebenfalls von der Änderung erfahren und konsistent bleiben.

Wir kennen das dem MVC-Schema zugrunde liegende Muster bereits aus Kapitel 9.4.3 unter dem Namen *Beobachter-Muster*. Mehrere Sichten beobachten ein Modell und werden benachrichtigt sobald es sich ändert. Das Besondere am MVC-Schema ist aber, daß das Modell nicht nur von Sichten, sondern auch von Eingabeteilen beobachtet wird. Alle Beobachter müssen sich beim Modell anmelden, damit dieses weiß, wer zu benachrichtigen ist.

Es ist wichtig, daß das Modell und seine Sichten in interaktiven Programmen getrennt werden. Faßt man sie zu einer einzigen Klasse zusammen, kann es immer nur *eine* Sicht auf ein Modell geben. Meist ist das eine unangenehme Einschränkung.

MVC-Schema als Framework

Wie bereits angedeutet, wird das MVC-Schema oft als Muster und nicht als konkretes Framework betrachtet. In manchen Bibliotheken liegt es aber auch als Framework vor und sieht dann wie in Abb. 11.8 beschrieben aus.

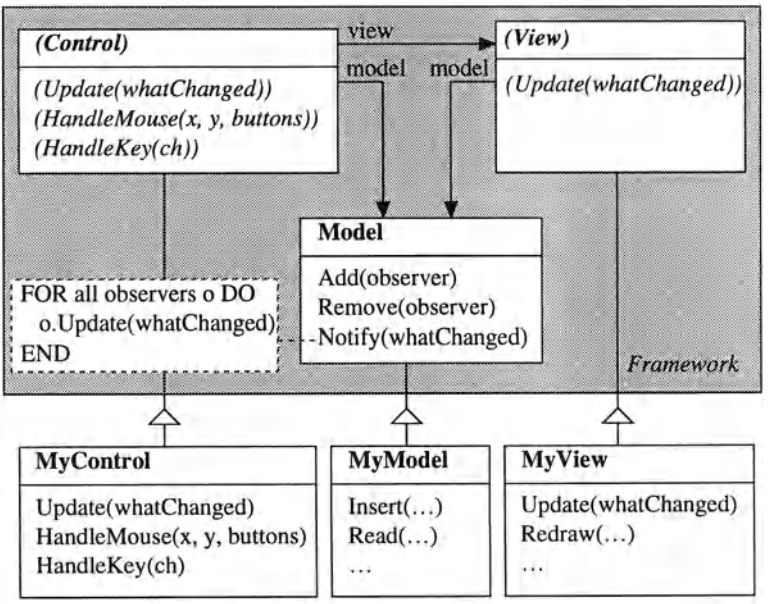


Abb. 11.8 Das MVC-Framework

Die Framework-Klassen implementieren schon eine gewisse Grundfunktionalität. Die Klasse *Model* hat zum Beispiel Methoden *Add* und *Remove*, mit denen sich Sichten und Eingabeteile an- und abmelden können. Sie hat außerdem eine Methode *Notify*, die von späteren Unterklassen immer dann aufgerufen werden sollte, wenn sich etwas am Modell ändert. Der geänderte Aspekt ist dabei im Objekt *whatChanged* codiert. Die *Notify*-Methode benachrichtigt dann alle Sichten und Eingabeteile von der Änderung, indem sie deren *Update*-Methode aufruft und ihnen den geänderten Aspekt mitteilt.

Um das Framework (den grauen Teil in Abb. 11.8) zu erweitern, leitet der Programmierer von jeder der drei Klassen eine Unterklasse ab. *MyControl* überschreibt zum Beispiel *HandleKey* so, daß eingetippte Zeichen mit *Insert* in *MyModel* eingefügt werden. *MyModel* enthält Methoden zum Ändern und Abfragen des Modells und ruft bei jeder Änderung *Notify* auf. *MyView* überschreibt die Methode *Update* zum Beispiel so, daß das geänderte Modell mit *Read* abgefragt und die Änderung mit *Redraw* am Bildschirm angezeigt wird.

MVC-Schema in Oberon

Das MVC-Schema wird auch im Oberon-System benutzt, allerdings in etwas anderer Implementierung als oben beschrieben. Sicht und Eingabeteil werden hier zu einer einzigen Klasse *Frame* zusammengefaßt (Abb. 11.9). Das ist vernünftig, weil diese beiden Teile ohnehin immer paarweise auftreten. Durch die Zusammenfassung redu-

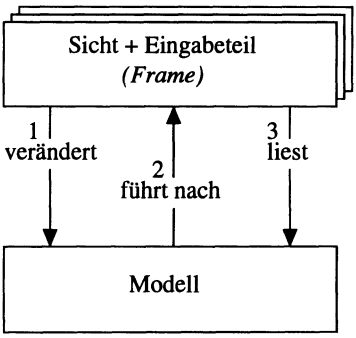


Abb. 11.9 MVC-Schema in Oberon

ziert sich nicht nur die Anzahl der Klassen, sondern auch die Anzahl der Meldungen zum Nachführen von Sichten und Eingabeteilen.

Im ursprünglichen MVC-Schema kennt das Modell die von ihm abhängigen Sichten und Eingabeteile: es verwaltet eine Liste dieser Objekte und schickt ihnen Meldungen, sobald es geändert wird. In der Implementierung des Oberon-Systems kennt das Modell seine Sichten *nicht*. Wenn das Modell geändert wird, schickt es eine Nachführ-Meldung an *alle* Sichten auf dem Bildschirm (broadcast). Die Sichten müssen selbst wissen, ob sie zum geänderten Modell gehören und daher auf die Meldung reagieren müssen oder nicht. Auf diese Weise werden zwar mehr Meldungen verschickt als nötig, aber die An- und Abmeldung der Sichten beim Modell entfällt.

Beispiel für das MVC-Schema

Wir wollen nun das Zusammenspiel zwischen Modell, Sicht und Eingabeteil im Oberon-System anhand eines Beispiels genauer betrachten: Angenommen, wir haben einen Texteditor vor uns. Sein Modell ist der editierte Text, der durch eine Klasse *Text* implementiert wird. Seine Sichten und Eingabeteile sind durch Textrahmen (Klasse *TextFrame*) dargestellt. Nehmen wir an, die Einfügemarke (*Caret*) steht in einem Textrahmen, und der Benutzer drückt eine Taste. Was geschieht? Das Oberon-System bestimmt den Rahmen, der die Einfügemarke enthält und schickt ihm die Meldung *HandleKey* zusammen mit dem Wert der gedrückten Taste. Das führt zum Aufruf folgender Methode:

```
PROCEDURE (f: TextFrame) HandleKey (ch: CHAR);
BEGIN
   IF ch = DEL THEN ... (*delete character to the left of the caret*)
   ELSE f.text.Write(ch) ... (*write ch to text*)
   END
END HandleKey;
```

Der Rahmen stellt *ch* also nicht gleich auf dem Bildschirm dar, sondern ändert lediglich das Modell (den Text). Der Text muß nun dafür sorgen, daß alle Rahmen nachgeführt werden, in denen er dargestellt wird. Er schickt dazu ein Meldungsobjekt von Typ *NotifyInsMsg* an das Fenstersystem (Modul *Viewers*), das es an alle Fenster verteilt und diese wiederum an alle in ihnen enthaltenen Rahmen.

```
TYPE
   NotifyInsMsg = RECORD (Message)
      t: Text;
      beg, end: LONGINT
   END;

PROCEDURE (t: Text) Write (ch: CHAR);
   VAR msg: NotifyInsMsg;
BEGIN
   ... (*insert ch at t.pos; t.pos := t.pos + 1*)
   msg.t := t; msg.beg := t.pos-1; msg.end := t.pos;
   Viewers.Broadcast(msg)
END Write;
```

Fenster und Rahmen haben einen *Meldungsinterpreter* im Sinne von Kapitel 9.4.1, der die Meldung zur Laufzeit untersucht und entweder darauf reagiert oder sie ignoriert. Der Meldungsinterpreter von Textrahmen sieht zum Beispiel so aus:

```
PROCEDURE (f: TextFrame) Handle (VAR m: Message);
BEGIN
   WITH
      m: Texts.NotifyInsMsg DO
         IF m.t = f.text THEN (* if the frame shows the modified text*)
            ... (*read m.t from m.beg to m.end*)
            ... (*and draw it on the screen*)
         END
   |  m: Texts.NotifyDelMsg DO
      ...
   ELSE (*ignore the message*)
   END
END Handle;
```

Nur wenn der Meldungsinterpreter *NotifyInsMsg* "versteht", und nur wenn der Rahmen das geänderte Modell anzeigt (d.h. *m.t = f.text*), wird die Änderung am Bildschirm nachgeführt. In allen anderen Fällen wird die Meldung ignoriert. Abb. 11.10 zeigt diesen Vorgang grafisch. Die grauen Rahmen sind diejenigen, die zum geänderten Modell gehören und auf *NotifyInsMsg* reagieren.

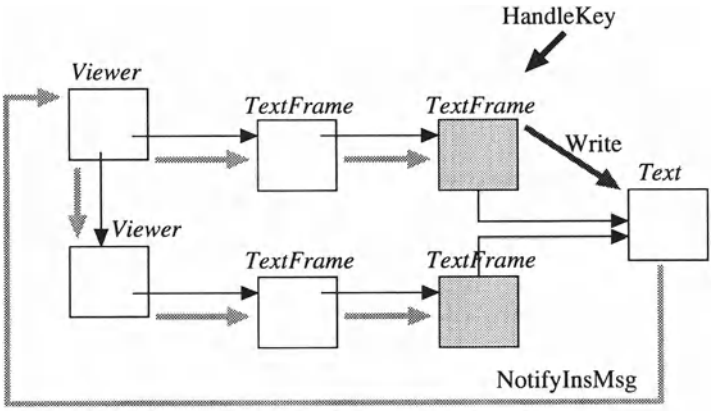


Abb. 11.10 Meldungsverteilung nach dem MVC-Schema

Wir haben hier ein schönes Beispiel dafür, wie ein und dieselbe Meldung an verschiedene Objekte verteilt wird. Man bezeichnet das im Englischen als *Broadcast*. Da der Sender die Empfänger nicht kennt, schickt er die Meldung einfach an alle. Nur diejenigen, für die die Meldung bestimmt ist, reagieren darauf. Eine Meldungsverteilung dieser Art ist ohne Meldungsobjekte kaum möglich.

11.4 Ein Framework für Objekte in Texten

In vielen Dokumenteneditoren kann ein Text nicht nur Zeichen enthalten, sondern auch andere Objekte wie Bilder, Tabellen oder Formeln, die im Text mitfließen (Abb. 11.11).

Der unter Oberon verfügbare Dokumenteneditor *Edit* [Szy92] beruht auf Texten dieser Art. Sie haben sich als äußerst nützlich und vielseitig erwiesen, vor allem, weil die Art der im Text enthaltenen Objekte nicht beschränkt ist. Der Programmierer kann neue Objektarten im-

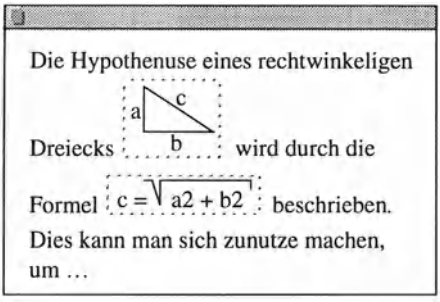


Abb. 11.11 Text mit mitfließenden Objekten

plementieren (z.B. Hypertext-Objekte) und sie wie andere Objekte im Text mitfließen lassen. Jedes Objekt behandelt Mausklicks auf seine Art: Bilder reagieren darauf, indem sie sich editieren lassen, Hypertext-Objekte, indem sie einen anderen Text anzeigen.

Elemente

Wir nennen im Text fließende Objekte *Elemente*. Zusammen mit Texten und Textrahmen ergeben sie ein Framework für viele nützliche Anwendungen, wie Dokumenteneditoren, Hypertext-Systeme, Tabellenkalkulationsprogramme oder allgemein Programme, die irgendwelche Objekte verwalten, anzeigen und editieren können.

Das Framework besteht in diesem Fall aus den Klassen *Text*, *TextFrame* und *Element*. Die Klassen *Text* und *TextFrame* sind konkret, während *Element* abstrakt ist (Abb. 11.12). Der Programmierer kann später daraus konkrete Elementarten wie Grafikelemente oder Formelelemente ableiten.

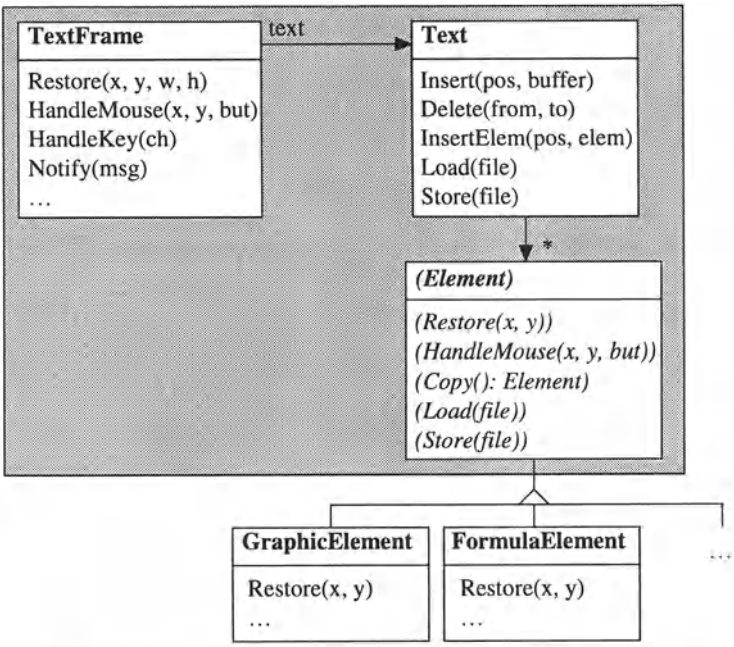


Abb. 11.12 Framework für Texte und mitfließende Elemente

Wie arbeiten Texte und Elemente zusammen? Ein Text verwaltet eine Liste von Elementen samt ihren Positionen. Wenn er von einer Datei geladen oder auf eine Datei gespeichert wird, müssen die Elemente mitgeladen oder mitgespeichert werden. Dazu schickt er ihnen *Load*- und *Store*-Meldungen, die jedes Element auf seine Weise interpretiert. Weiter muß ein Text nichts über seine Elemente wissen.

Welche Operationen führt ein Textrahmen mit Elementen aus? Wenn der Inhalt des Rahmens neu gezeichnet wird, bekommen die Elemente eine *Restore*-Meldung, die sie veranlaßt, sich an einer bestimmten Position im Rahmen darzustellen. Wenn der Benutzer mit der Maus auf ein Element zeigt, schickt ihm der Rahmen eine *HandleMouse*-Meldung, auf die das Element reagieren kann. Wenn schließlich ein Textstück kopiert werden soll, bekommen alle darin enthaltenen Elemente eine *Copy*-Meldung. Ein Textrahmen muß nicht wissen, *welche* Elementarten es gibt. Er kommuniziert mit Elementen nur über Meldungen und läßt so *beliebige* Elementarten zu.

Das Framework kann durch konkrete Element-Klassen ausgebaut werden, etwa durch *GraphicElement*, *FormulaElement* oder *TableElement*. Diese Klassen sind dem Editor nicht bekannt. Während der Editor läuft, können sie dynamisch dazugeladen und in den Text eingefügt werden (siehe Kapitel 9.4.7). Sie erweitern die Mächtigkeit des Editors ganz nach den Bedürfnissen des Benutzers.

Man kann nicht genug betonen, wie wichtig die Möglichkeit ist, Erweiterungen *zur Laufzeit* hinzufügen zu können. Erst das macht Programme *jederzeit* erweiterbar, ohne sie neu übersetzen oder binden zu müssen.

Der Leser möge diesen Ansatz mit anderen ihm bekannten Editoren vergleichen. Viele Editoren müssen immer mit ihrer gesamten Funktionalität geladen (oder zumindest gebunden) werden; das führt zu langen Ladezeiten und großem Speicherbedarf und überhäuft den Benutzer mit einer Fülle von Funktionen, die er meist nie benutzt. Durch die Erweiterbarkeit von Oberon-Programmen zur Laufzeit hat jeder Benutzer nur den *Kern* des Editors und diejenigen Funktionen im Speicher, die er auch braucht.

Kapitel 12 enthält eine vollständige Implementierung von Texten mit Elementen.

11.5 Application Frameworks

Wenn man aus den Gemeinsamkeiten von *Teilsystemen* ein Framework herausziehen kann, warum sollte es dann nicht möglich sein, aus *ganzen Programmen* die gemeinsamen Teile herauszuziehen? In *älteren* Programmen gibt es kaum solche gemeinsamen Teile. Moderne, ereignisgesteuerte Programme ähneln einander jedoch tatsächlich und man findet Gemeinsamkeiten, die zu einem sogenannten *Application Framework* zusammengefaßt werden können.

Dialogprogramme

Dialogprogramme der ersten Generation erfragen üblicherweise die Eingabedaten in einer *festgelegten Reihenfolge*. Falsche Eingaben können oft nicht rückgängig gemacht werden, weil das Programm

bereits die nächste Eingabe verlangt. Das Programm *gängelt* den Benutzer: er kann Eingaben nicht in beliebiger Reihenfolge vornehmen.

Die nächste Generation von Dialogprogrammen verwendet Menüs, die Eingaben in *beliebieger Reihenfolge* zulassen. Allerdings werden Menüs oft hierarchisch gegliedert, mit einem Hauptmenü und mehreren Untermenüs, die wiederum Untermenüs enthalten können. Jedem Untermenü entspricht ein Programmzweig (Abb. 11.13).

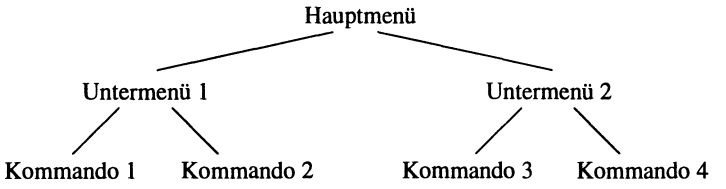


Abb. 11.13 Hierarchische Menüs führen Zustände ein

Der Benutzer kann jetzt zwar Kommandos in beliebiger Reihenfolge ausführen, muß aber im Menübaum auf und ab steigen, bis er in den richtigen Zustand (Modus) gelangt, in dem das gewünschte Kommando erlaubt ist. Modi, in denen nur bestimmte Eingaben erlaubt sind, beeinträchtigen die Benutzerfreundlichkeit eines Programms und sind wenn möglich zu vermeiden.

Ereignisgesteuerte Dialogprogramme

Moderne Dialogprogramme sind *ereignisgesteuert*. Sie haben nur einen *einzigen* Zustand, in dem *alle* Eingaben in *beliebiger Reihenfolge* möglich sind. Jede Eingabe (Tastendruck, Mausklick, Menükommando, usw.) ist ein Ereignis und bewirkt den Aufruf einer entsprechenden Behandlungsroutine oder eine Meldung an ein Objekt. Der Kern des Programms ist also eine Schleife, die Eingaben entgegennimmt und verteilt (Abb. 11.14). Die Programmstruktur ist invertiert und erinnert an das Hollywood-Prinzip: "don't call us, we'll call you". Im Oberon-System liegt diese Schleife im Kernmodul *Oberon*. Sie muß nicht in jedem Programm neu implementiert werden.

Diese Steuerlogik ist wiederverwendbar. Sie ist allen Dialogprogrammen gemeinsam und daher Bestandteil des Frameworks. Dialogprogramme haben noch andere Gemeinsamkeiten: Sie arbeiten häufig mit Fenstern und können diese auf einheitliche Weise verschieben, vergrößern und verkleinern. Diese Operationen sind unabhängig vom Fensterinhalt und können daher ins Framework übernommen werden. Weitere gemeinsame Teile sind Zeichenflächen (Rahmen), Dialogknöpfe oder Menüs.

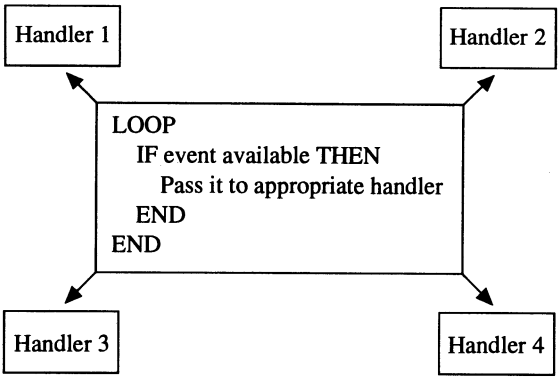


Abb. 11.14 Ereignisgesteuerte Programmstruktur

Application Frameworks

Ein *Application Framework* ist ein Standardprogramm, das die Grundfunktionalität aufweist, die man von *jeder* Anwendung erwartet: Fensterverwaltung (noch ohne Fensterinhalt), Menüs, Meldungen zum Laden und Speichern von Dokumenten, Abfangen von Mausklicks und Tastatureingaben (noch ohne sinnvolle Verarbeitung) usw. Diese Funktionalität wird durch ein Geflecht von Klassen erreicht, die zum Teil konkret sind und somit fertiges Verhalten implementieren, zum Teil aber auch abstrakt und erst durch Unterklassen konkretisiert werden müssen. Ein Application Framework ist ein lauffähiges Programm, das bereits mit Fenstern und Menüs umgehen kann, aber erst vom Programmierer erweitert werden muß, um den Inhalt der Fenster und Menüs zu füllen.

Es gibt verschiedene Application Frameworks, die das Schreiben von Dialogprogrammen erheblich erleichtern. Einige der bekannteren sind *MacApp* [Sch86], *NextStep* [Web89] und *ET++* [GWM88].

MacApp

Als Beispiel für eines dieser Application Frameworks greifen wir *MacApp* heraus: MacApp wird von der Firma *Apple* angeboten. Es ist in *Object-Pascal* implementiert und enthält eine Bibliothek von Klassen, die auf bestimmte Weise zu einem Framework zusammengesetzt sind (Abb. 11.15).

Jedes MacApp-Programm ist eine Erweiterung der Klasse *Application*. Eine Applikation bearbeitet ein oder mehrere Dokumente, die in einem oder mehreren Fenstern dargestellt werden. Ein Fenster enthält eine oder mehrere Zeichenflächen (*View*), die Text, Grafik oder sonstige Daten darstellen und auf Benutzereingaben reagieren. In MacApp wird eine etwas andere Terminologie verwendet als in Oberon. *Window* entspricht in Oberon einem *Viewer*, *View* einem *Frame*.

Jede Klasse des MacApp-Frameworks erfüllt bereits eine gewisse Aufgabe. *Application* erledigt Initialisierungsarbeiten und verteilt Ereignisse, *Window* übernimmt das Verschieben und Verändern der Fen-

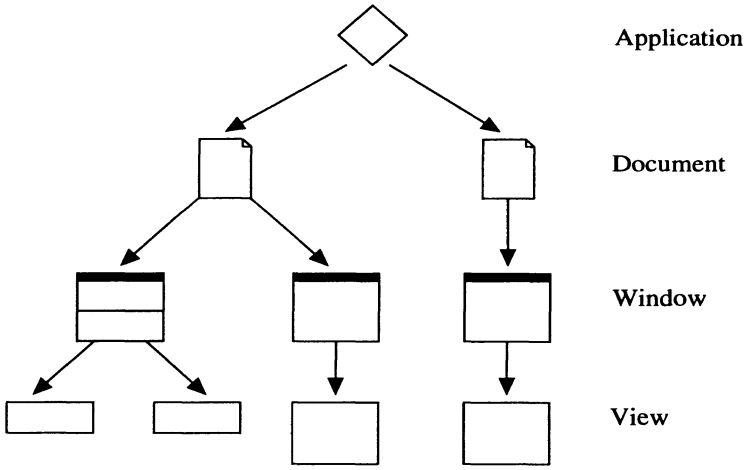


Abb. 11.15 Application Framework in *MacApp*. Pfeile deuten die Benutzt-Beziehung an

ster und View *das* Anpassen der Zeichenflächengröße an das umgebende Fenster.

Durch Erweitern dieser Klassen und Überschreiben einiger ihrer Methoden kann man aus dem Standardverhalten des Frameworks eine konkrete Anwendung machen. Das ist ein schönes Beispiel für *programming by difference*: man programmiert nur diejenigen Teile, die vom Standardverhalten abweichen.

12 Fallstudie Oberon0

In manchen Büchern wird objektorientierte Programmierung anhand kleiner Beispiele für Stacks, Listen oder Queues gelehrt. Diese Beispiele sind alles andere als repräsentativ, ja sie vermitteln ein völlig falsches Bild von den wirklichen Anwendungen und Vorteilen dieser Technik. Objektorientiertes Programmieren ist "Programmieren im Großen" und erfordert große, realistische Beispiele. Daher soll nun eine realistische Fallstudie präsentiert werden: Ein System von Fenstern, die man verschieben, vergrößern und verkleinern kann, in denen man Text und Grafik editieren kann – und das Ganze im Quellcode.

Durch das Lesen von Quellcode kann man viel lernen. In der Schule lernen wir *zuerst* Lesen und dann erst Schreiben. Wir lesen gute Bücher, um unseren Stil zu verbessern. Warum studieren wir dann so wenig Programme? Warum versuchen wir nicht, deren Stil in uns aufzunehmen, bevor wir eigene Programme schreiben? Vielleicht weil so wenig Programme im Quellcode veröffentlicht werden. Wo Quellcode in ansprechender Form zugänglich ist, wird er meist auch dankbar gelesen.

Oberon versus Oberon0

Das beschriebene System trägt den Namen *Oberon0*, weil es sich in der Funktionalität und Implementierung stark an Oberon [WiG92] anlehnt. Einiges wurde jedoch anders implementiert: Die meisten Meldungen wurden als Methoden und nicht wie in Oberon als Meldungsobjekte implementiert. Details von Oberon, die den Quellcode vergrößert hätten, ohne etwas zur objektorientierten Idee beizutragen, wurden weggelassen. Oberon0 ist also weniger mächtig als Oberon und auch nicht so effizient. Es ist aber ein realistisches und funktionsfähiges System, das man für einfache Editieraufgaben benutzen kann. Der Quellcode von Oberon0 ist zusammen mit dem Oberon-System auf der Begleit-CD zu diesem Buch enthalten (siehe auch Anhang D).

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

Oberon0 besteht aus fünf Teilen mit den folgenden Aufgaben:

1. Verwaltung von Fenstern und Zeichenflächen
2. Verteilung der Benutzereingaben
3. Editieren von Texten
4. Editieren von Grafiken
5. Einbettung von Grafiken in Texte

Das System besteht aus 1300 Zeilen Quellcode, 11 Modulen und 11 Klassen. Jedes Modul und jede Klasse werden zuerst allgemein beschrieben, dann folgt eine annotierte Programmliste mit Erläuterungen. Die in der Fallstudie importierten Bibliotheksmodule wie *OS*, *In* und *Out* sind in Anhang B beschrieben.

Der Leser wird feststellen, daß ein großer Teil von Oberon0 in konventionellem Stil geschrieben ist: nicht alle Datentypen sind Klassen; nicht alle Operationen sind Methoden. Das ist kein Mangel, sondern eine bewußte Entwurfsentscheidung. Klassen werden nur dort eingesetzt, wo sie das Programm einfacher oder erweiterbar machen. Es ist ein Ziel dieser Fallstudie, dem Leser zu zeigen, wo man Klassen sinnvoll einsetzt, und wo man lieber auf sie verzichtet.

Der Leser möge sich für dieses Kapitel Zeit nehmen. Man kann es nicht als Bettlektüre verschlingen, sondern muß es mit Papier und Bleistift in der Hand studieren. Nur durch das Studium vollständiger Beispiele bekommt man die nötige Erfahrung, um selbst objektorientierte Programme zu schreiben.

12.1 Das Fenstersystem

Das Fenstersystem von Oberon0 verwaltet rechteckige Bereiche auf einem Rasterbildschirm, in denen man Daten betrachten und editieren kann. Man nennt diese Bereiche *Fenster* oder *Viewer*.

Fenster

Fenster unterteilen den Bildschirm vollständig in Rechtecke (*tiling*

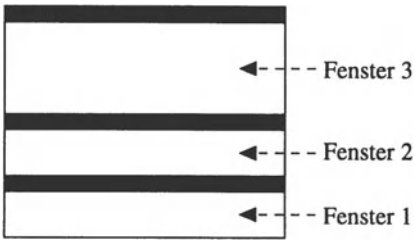


Abb. 12.1 Oberon0-Bildschirm mit drei Fenstern

viewers). Der Einfachheit halber gibt es in Oberon0 nur *eine* Fensterleiste (Abb. 12.1) und nicht *zwei* wie in Oberon.

Die schwarzen Balken am oberen Rand der Fenster sind die *Titelbalken*, in denen der Name des Fensters steht. Wenn man in ihnen den linken Mausknopf drückt, kann man mit der Maus den oberen Rand des Fensters nach oben und unten verschieben und das Fenster dadurch vergrößern oder verkleinern. Ferner gibt es Kommandos, um ein Fenster zu öffnen und zu schließen.

Rahmen

Um Daten darzustellen, zeichnet man nicht direkt in ein Fenster, sondern in einen rechteckigen *Zeichenrahmen* (*Frame*), den man im Fenster anbringt. Rahmen haben zwei Aufgaben:

1. Man kann in ihnen Text oder Grafik darstellen.
2. Sie reagieren auf Benutzereingaben (Mausklicks und Tastatureingaben).

Genau die gleichen Aufgaben hat auch ein Fenster: es ist dafür zuständig, seinen Rand zu zeichnen und Benutzereingaben zu interpretieren, die es allerdings meist an die in ihm enthaltenen Rahmen weiterleitet. Ein Fenster ist also selbst eine Art Rahmen und daher eine Unterklasse von *Frame*. Der Einfachheit halber enthält ein Fenster in Oberon0 immer genau zwei Rahmen: einen *Menürahmen*, der den Namen des Fensters und eine Liste von Oberon-Kommandos enthält, und einen *Inhaltsrahmen*, in dem die eigentlichen Daten (Text oder Grafik) erscheinen (Abb. 12.2). Ein Rahmen vereinigt in sich die Aufgaben der Sicht und des Eingabeteils aus dem MVC-Schema (Kapitel 11.3).

Fenster und Rahmen sind so eng miteinander verwandt, daß es sinnvoll ist, sie in ein gemeinsames Modul *Viewers0* zu verpacken, dessen Schnittstelle folgendermaßen aussieht (Alle Module dieser Fallstudie haben einen Namen mit einer angehängten "0", die sie von den gleichnamigen Modulen des Oberon-Systems unterscheidet):

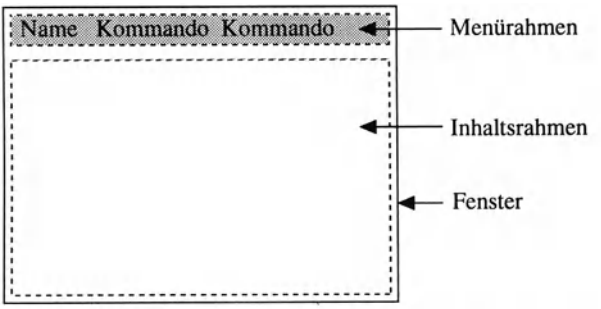


Abb. 12.2 Oberon0-Fenster mit Menürahmen und Inhaltsrahmen

Schnittstelle von Viewers0

```
DEFINITION Viewers0;
IMPORT OS;

TYPE
   Frame = POINTER TO FrameDesc;
   FrameDesc = RECORD (OS.ObjectDesc)
      x, y: INTEGER;  (*left bottom in pixels relative to left bot. of screen*)
      w, h: INTEGER;  (*width, height in pixels*)
      PROCEDURE (f: Frame) Draw;
      PROCEDURE (f: Frame) Modify (dy: INTEGER);
      PROCEDURE (f: Frame) Move (dy: INTEGER);
      PROCEDURE (f: Frame) Copy (): Frame;
      PROCEDURE (f: Frame) HandleKey (ch: CHAR);
      PROCEDURE (f: Frame) HandleMouse (x, y: INTEGER; but: SET);
      PROCEDURE (f: Frame) Handle (VAR m: OS.Message);
      PROCEDURE (f: Frame) Neutralize;
      PROCEDURE (f: Frame) SetFocus;
      PROCEDURE (f: Frame) Defocus;
   END;

   Viewer = POINTER TO ViewerDesc;
   ViewerDesc = RECORD (FrameDesc)
      menu-, cont-: Frame;
      next-: Viewer;
      PROCEDURE (v: Viewer) Close;
   END;

VAR
   focus-: Frame;   (*the frame that gets the keyboard input*)

PROCEDURE New (menu, cont: Frame): Viewer;
PROCEDURE ViewerAt (y: INTEGER): Viewer;
PROCEDURE Broadcast (VAR m: OS.Message);

(*commands*)
PROCEDURE Close;
PROCEDURE Copy;

END Viewers0.
```

Die Position und Größe eines Rahmens *f* wird in Abb. 12.3 dargestellt. Die Koordinaten (*f.x*, *f.y*) sind relativ zur linken unteren Ecke des Bildschirms.

Frame ist eine *abstrakte* Klasse. Sie gibt nur eine Schnittstelle vor, ohne sie vollständig zu implementieren. Aufgrund der *Frame*-Schnittstelle weiß ein Fenster aber, welche Operationen es mit einem Rahmen ausführen kann. Und da ein Fenster mit Rahmen arbeiten kann, kann es auch mit jeder Erweiterung davon arbeiten, zum Beispiel mit Textrahmen (Kapitel 12.3.3) oder mit Grafikrahmen (Kapitel 12.4.2).

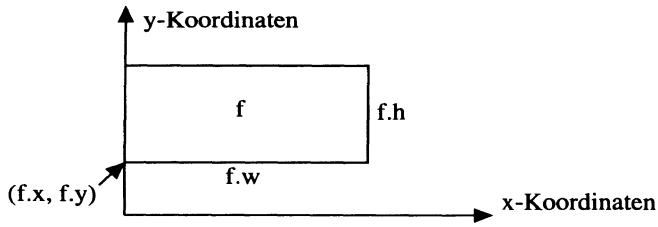


Abb. 12.3 Position und Größe eines Rahmens *f*

Frame-Meldungen

- *f.Draw* fordert den Rahmen *f* auf, seinen Inhalt neu zu zeichnen.
- *f.Modify(dy)* verschiebt den unteren Rand des Rahmens *f* um *dy* Punkte nach oben ($dy > 0$) oder nach unten ($dy < 0$).
- *f.Move(dy)* verschiebt den gesamten Rahmen *f* um *dy* Punkte nach oben ($dy > 0$) oder nach unten ($dy < 0$).
- *f1 := f.Copy()* liefert eine Kopie des Rahmens *f*.
- *f.HandleKey(ch)* fordert den Rahmen *f* auf, das Zeichen *ch* (von der Tastatur) zu verarbeiten. Diese Meldung wird einem Rahmen nur dann geschickt, wenn er der *Focus-Rahmen* ist (siehe später).
- *f.HandleMouse(x, y, b)* fordert den Rahmen *f* auf, auf die Maus zu reagieren. Diese Meldung wird dem Rahmen mehrmals pro Sekunde geschickt, solange er den Mauszeiger enthält. *x* und *y* sind die Mauskoordinaten relativ zum linken unteren Rand des Bildschirms und *b* ist die Menge der gedrückten Mausknöpfe.
- *f.Handle(m)* analysiert das Meldungsobjekt *m* und reagiert darauf (*Handle* ist der Meldungsinterpreter von *Frame*).
- *f.Defocus* wird dem Focus-Rahmen *f* geschickt, unmittelbar bevor ein anderer Rahmen zum Focus wird. Der Rahmen sollte als Reaktion darauf seine Markierungen (z.B. Selektion) entfernen.
- *f.SetFocus* macht *f* zum Focus-Rahmen.
- *f.Neutralize* fordert den Rahmen *f* auf, alle seine Markierungen (Caret, Selektion, usw.) zu entfernen.

Viewer-Meldungen

Ein Fenster erbt die Schnittstelle von Rahmen, überschreibt aber einige Methoden. Zum Beispiel muß beim Vergrößern und Verkleinern eines Fensters ein Teil seines Randes neu gezeichnet werden. Zusätzlich verstehen Fenster eine Meldung *Close*.

- *v.Close* fordert das Fenster *v* auf, sich zu schließen.

Einer der Rahmen auf dem gesamten Bildschirm ist der sogenannte *Focus-Rahmen*. An ihn werden mittels *HandleKey* alle Zeichen geschickt, die von der Tastatur eingegeben werden.

Die Prozedur *New* erzeugt ein Fenster und zeigt es am Bildschirm an. *ViewerAt(y)* liefert das Fenster, in dem die Koordinate *y* liegt. *Broadcast(m)* schickt das Meldungsobjekt *m* an alle Fenster auf dem Bildschirm.

Copy und *Close* sind Oberon-Kommandos, die durch Anklicken am Bildschirm aktiviert werden. *Close* schließt das Fenster, das den Kommandonamen enthält. *Copy* erzeugt eine Kopie von ihm und zeigt sie am Bildschirm an.

Alle Fenster sind durch ein Feld *next* miteinander verkettet. Eine globale Variable *viewers* zeigt auf das unterste Fenster am Bildschirm (Abb. 12.4).

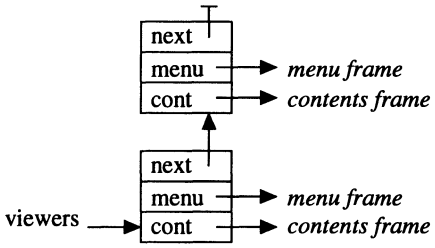


Abb. 12.4 Liste aller Fenster am Bildschirm

Es folgt nun die Implementierung des Moduls *Viewers0*. Stellen, die einer Erläuterung bedürfen, sind am Rand mit einer Nummer versehen. Unter dieser Nummer finden sich im Anschluß an den Code Kommentare. Das importierte Modul *OS* ist in Anhang B beschrieben.

Viewers0

```
MODULE Viewers0;
IMPORT OS;

TYPE
   Frame* = POINTER TO FrameDesc;
   FrameDesc* = RECORD (OS.ObjectDesc)
      x*, y*: INTEGER;   (*left bottom in pixels relative to left bot. of screen*)
      w*, h*: INTEGER    (*width, height in pixels*)
   END;
   Viewer* = POINTER TO ViewerDesc;
   ViewerDesc* = RECORD (FrameDesc)
      menu-, cont-: Frame;        (*menu frame, contents frame*)
      next-: Viewer;
   END;

VAR
   focus-: Frame;      (*the frame that gets the keyboard input*)
   viewers: Viewer;    (*root for list of viewers on the screen*)
   barH: INTEGER;      (*default height of title bar*)
   minH: INTEGER;      (*minimal height of a viewer*)
```

Frame-Methoden

```
PROCEDURE (f: Frame) Draw*;
BEGIN HALT(99) (*abstract*)
END Draw;

PROCEDURE (f: Frame) Copy* (): Frame;
BEGIN HALT(99) (*abstract*)
END Copy;

PROCEDURE (f: Frame) Neutralize*;
BEGIN HALT(99) (*abstract*)
END Neutralize;

PROCEDURE (f: Frame) HandleKey* (ch: CHAR);
BEGIN HALT(99) (*abstract*)
END HandleKey;

PROCEDURE (f: Frame) HandleMouse* (x, y: INTEGER; buttons: SET);
BEGIN HALT(99) (*abstract*)
END HandleMouse;

PROCEDURE (f: Frame) Handle* (VAR m: Message);
BEGIN HALT(99) (*abstract*)
END Handle;

PROCEDURE (f: Frame) Modify* (dy: INTEGER);
BEGIN
   INC(f.y, dy); DEC(f.h, dy)
END Modify;

PROCEDURE (f: Frame) Move* (dy: INTEGER);
BEGIN
   INC(f.y, dy)
END Move;

PROCEDURE (f: Frame) Defocus*;
BEGIN
   focus := NIL
END Defocus;

PROCEDURE (f: Frame) SetFocus*;
BEGIN
   IF focus # NIL THEN focus.Defocus END; focus := f
END SetFocus;
```

Viewer-Methoden

```
PROCEDURE (v: Viewer) Erase (h: INTEGER);
BEGIN
   IF h > 0 THEN (*clear bottom block and draw left and right border*)
      OS.EraseBlock(v.x, v.y, v.w, h);
      OS.FillBlock(v.x, v.y, 1, h);
      OS.FillBlock(v.x+v.w-1, v.y, 1, h)
   END;
   OS.FillBlock(v.x, v.y, OS.screenW, 1)
END Erase;
```

```
PROCEDURE (v: Viewer) FlipTitleBar;
BEGIN
   OS.InvertBlock(v.x+1, v.y+v.h-barH, OS.screenW-2, barH)
END FlipTitleBar;

PROCEDURE (v: Viewer) Neutralize*;
BEGIN
   v.menu.Neutralize; v.cont.Neutralize
END Neutralize;

PROCEDURE (v: Viewer) Modify* (dy: INTEGER);
BEGIN
   v.Neutralize;
   v.Modify^ (dy); v.Erase(-dy+1); v.cont.Modify(dy)
END Modify;

PROCEDURE (v: Viewer) Move* (dy: INTEGER);
BEGIN
   v.Neutralize;
   v.menu.Move(dy); v.cont.Move(dy);
   OS.CopyBlock(v.x, v.y+1, v.w, v.h-1, v.x, v.y+dy+1);
   INC(v.y, dy)
END Move;

PROCEDURE (v: Viewer) Draw*;
BEGIN
   OS.FadeCursor;
   v.Erase(v.h); v.menu.Draw; v.cont.Draw; v.FlipTitleBar
END Draw;

PROCEDURE (v: Viewer) HandleMouse* (x, y: INTEGER; buttons: SET);
   VAR b: SET; x1, y1: INTEGER; dy, maxUp, maxDown: INTEGER;
BEGIN
   OS.DrawCursor(x, y);
   IF y > v.menu.y THEN
      IF OS.left IN buttons THEN (*left click in menu bar => resize viewer*)
         (*----- track mouse movements*)
         v.FlipTitleBar;
         REPEAT
            OS.GetMouse(b, x1, y1); OS.DrawCursor(x1, y1)
         UNTIL b = {};
         v.FlipTitleBar;
1 ☞      (*----- compute how far v can be moved up or down*)
         dy := y1 - y; maxDown := v.h - minH;
         IF v.next = NIL THEN maxUp := OS.screenH - v.y - v.h
         ELSE maxUp := v.next.h - minH; v.next.Neutralize
         END;
         IF dy < - maxDown THEN dy := - maxDown
         ELSIF dy > maxUp THEN dy := maxUp
         END;
         (*----- move v up or down and adjust neighbour viewers*)
         OS.FadeCursor; v.Neutralize;
         IF dy < 0 THEN (*move down*) v.Modify(-dy); v.Move(dy)
```

```
            ELSE (*move up*) v.Move(dy); v.Modify(-dy)
            END;
            IF v.next # NIL THEN v.next.Modify(dy)
            ELSE OS.EraseBlock(v.x, v.y+v.h, v.w, OS.screenH-v.y-v.h)
            END
        ELSE v.menu.HandleMouse(x, y, buttons)
        END
    ELSE v.cont.HandleMouse(x, y, buttons)
    END
END HandleMouse;

PROCEDURE (v: Viewer) Handle* (VAR m: OS.Message);
BEGIN
    v.menu.Handle(m); v.cont.Handle(m)
END Handle;

PROCEDURE (v: Viewer) Close*;
    VAR x: Viewer;
BEGIN
    OS.FadeCursor; v.Neutralize;
    IF v.next # NIL THEN v.next.Modify(-v.h)
    ELSE OS.EraseBlock(v.x, v.y, v.w, v.h)
    END;
    IF viewers = v THEN viewers := v.next
    ELSE
        x := viewers;
        WHILE x.next # v DO x := x.next END;
        x.next := v.next
    END
END Close;

PROCEDURE ViewerAt*(y: INTEGER): Viewer;
    VAR v: Viewer;
BEGIN
    v := viewers;
    WHILE (v # NIL) & (y > v.y + v.h) DO v := v.next END;
    RETURN v
END ViewerAt;
```

Sonstige Prozeduren

```
PROCEDURE New* (menu, cont: Frame): Viewer;
    VAR below, above, v, w: Viewer; top: INTEGER;
BEGIN
    (*----- compute position of new viewer*)
    IF ViewerAt(OS.screenH) = NIL THEN
        top := OS.screenH
    ELSE
        w := viewers; v := viewers.next;
        WHILE v # NIL DO
            IF v.h > w.h THEN w := v END;
            v := v.next
        END;
        top := w.y + w.h DIV 2
    END;
```

2 ☞

```
    (*----- generate ne viewer and link it into viewer list*)
    above := viewers; below := NIL;
    WHILE (above # NIL) & (top > above.y + above.h) DO
        below := above; above := above.next
    END;
    NEW(v); v.x := 0; v.w := OS.screenW; v.next := above;
    IF below = NIL THEN v.y := 0; v.h := top
    ELSE v.y := below.y + below.h; v.h := top - v.y
    END;
    IF v.h < minH THEN RETURN NIL END;
    v.menu := menu; v.cont := cont;
    menu.x := v.x+1; menu.y := v.y+v.h-barH;
    menu.w := v.w-2; menu.h := barH-1;
    cont.x := v.x+1; cont.y := v.y+1;
    cont.w := v.w-2; cont.h := menu.y - v.y-1;
    IF below = NIL THEN viewers := v ELSE below.next := v END;
    IF above # NIL THEN above.Modify(v.h) END;
    v.Draw;
    RETURN v
END New;

PROCEDURE Broadcast* (VAR m: OS.Message);
    VAR v: Viewer;
BEGIN
    v := viewers;
    WHILE v # NIL DO v.Handle(m); v := v.next END
END Broadcast;

PROCEDURE Init;
    VAR f: OS.Font;
BEGIN
    viewers := NIL; focus := NIL;
    f := OS.DefaultFont();
    barH := f.height + 2; minH := barH + 2
END Init;
```

Kommandos

```
PROCEDURE Close*;
    VAR x, y: INTEGER; buttons: SET; v: Viewer;
BEGIN
    OS.GetMouse(buttons, x, y); v := ViewerAt(y); v.Close
END Close;

PROCEDURE Copy*;
    VAR v: Viewer; x, y: INTEGER; buttons: SET;
BEGIN
    OS.GetMouse(buttons, x, y); v := ViewerAt(y);
    v := New(v.menu.Copy(), v.cont.Copy())
END Copy;

BEGIN
    Init
END Viewers0.
```

Die meisten *Frame*-Methoden sind abstrakt und müssen in Unterklassen überschrieben werden: Für abstrakte Rahmen kann man eben noch nicht angeben, welchen Inhalt sie darstellen sollen oder wie sie auf Mausklicks und Tastatureingaben reagieren sollen. Die Methoden *Move*, *SetFocus* und *Defocus* können jedoch bereits für abstrakte Rahmen sinnvoll implementiert werden, was daher auch in der Klasse *Frame* geschehen ist. Sie müssen sogar in den meisten Fällen gar nicht mehr in Unterklassen überschrieben werden. *Modify* ist ebenfalls bereits in *Frame* implementiert, muß aber in Unterklassen so überschrieben werden, daß der nach einer Veränderung sichtbar gewordene Inhalt des Rahmens angezeigt wird (siehe zum Beispiel Kapitel 12.3.3).

Erläuterungen

Ein Fenster reagiert auf einen Mausklick in den Titelbalken, indem es seinen oberen Rand verschiebt. Dieser Vorgang bedarf einer Erklärung (Abb. 12.5). *1* ☞

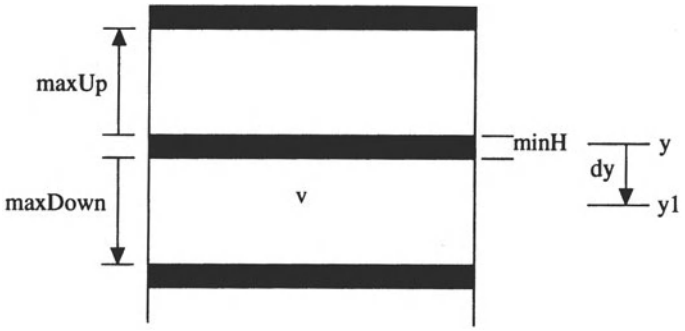


Abb. 12.5 Verschieben des Fensters *v* um *dy* Punkte

Die Maus wurde an der Position *y* gedrückt und bei *y1* losgelassen. Der Verschiebungsvektor in *y*-Richtung ist also *dy* = *y1-y*. Da der Titelbalken eines Fensters immer sichtbar bleiben muß, kann der Fensterrand höchstens um *maxUp* Punkte nach oben oder um *maxDown* Punkte nach unten verschoben werden, bis er an das Nachbarfenster stößt. Fenster können dadurch eine gewisse Minimalhöhe *minH* nicht unterschreiten. Durch das Verschieben des Fensterrandes wird das darüber liegende Fenster größer oder kleiner und bekommt daher eine *Modify*-Meldung. Wenn oberhalb von v kein Fenster mehr liegt, ist maxup die Distanz zwischen v und dem oberen Bildschirmrand.

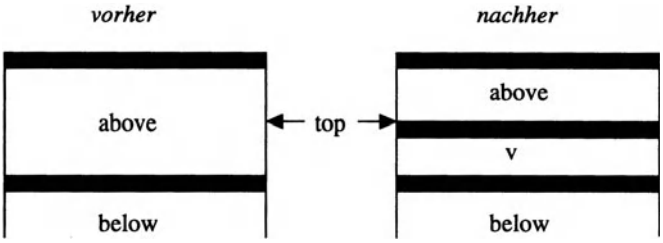


Abb. 12.6 Öffnen eines Fensters mit oberem Rand *top*

2 ☞

Die Position eines neuen Fensters wird so festgelegt, daß sein oberer Rand *top* entweder ganz oben am Bildschirm liegt (falls sich dort noch kein anderes Fenster befindet) oder in der Mitte des Fensters mit der größten Höhe. In letzteren Fall kommt das neue Fenster *v* zwischen zwei anderen Fenstern *below* und *above* zu liegen (Abb. 12.6). Das Fenster *above* wird dadurch verkleinert und bekommt daher eine *Modify*-Meldung.

Was kann man aus dieser Implementierung lernen?

Viewer und *Frame* sind zwei Bausteine mit einigermaßen komplexen Daten und nützlichen Operationen. Es ist daher gerechtfertigt, sie als Klassen zu implementieren. Da vorauszusehen ist, daß in Fenstern

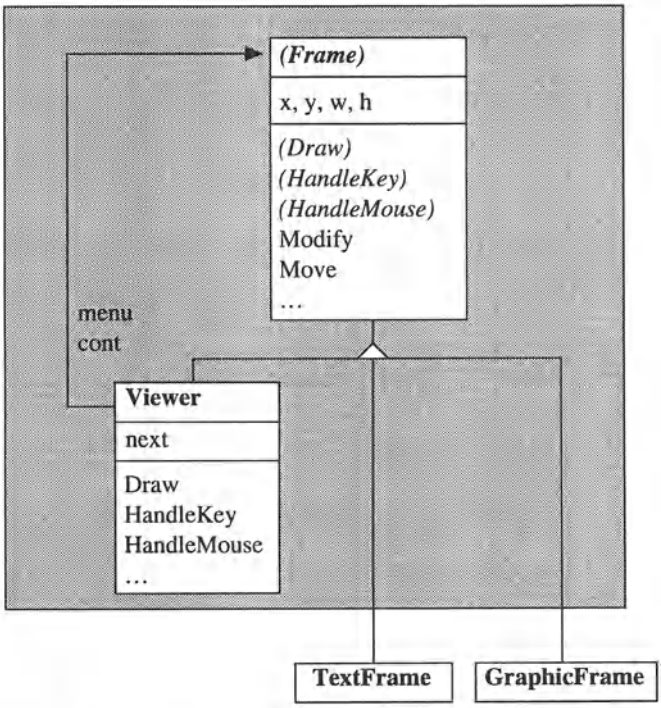


Abb. 12.7 Framework aus Rahmen und Fenstern

verschiedene Varianten von Rahmen angezeigt werden (Textrahmen, Grafikrahmen, etc.), ist es sinnvoll, das gemeinsame Verhalten aller Rahmen in einer abstrakten Klasse *Frame* zu definieren. Fenster arbeiten nicht mit den konkreten Rahmen-Varianten, sondern mit abstrakten Rahmen, daher brauchen sie diese Varianten nicht zu kennen. Sie sind in der Lage, jede beliebige Rahmen-Variante, die in Zukunft einmal entwickelt wird, anzuzeigen, ohne daß man die Implementierung der Fenster ändern muß. Wir haben hier ein Beispiel für heterogene, erweiterbare Datenstrukturen (Kapitel 8.3). Man kann die Klassen *Viewer* und *Frame* auch als Framework eines erweiterbaren Fenstersystems betrachten (Abb. 12.7). Der Kern des Frameworks entspricht dem *Kompositum*-Muster aus Kapitel 9.3.3.

Obwohl es möglich wäre, auch Fenster später einmal durch eine Unterklasse zu erweitern, ist dies nicht vorgesehen und im allgemeinen auch nicht nötig. Fenster sind in dieser Implementierung immer gleichbleibende Behälter für Rahmen. Nur ihr *Inhalt* soll variieren, die Fenster selbst nicht.

12.2 Verteilung von Benutzereingaben

Fenster und Rahmen können auf Tastatureingaben und Mausklicks reagieren. Aber wer meldet ihnen diese Ereignisse? Das ist Aufgabe der sogenannten *Hauptschleife* (*main event loop*). Wann immer das System nichts anderes zu tun hat, befindet es sich in der Hauptschleife und prüft den Zustand der Eingabequellen.

Tastatureingaben

Wenn man eine Taste drückt, wird das entsprechende Zeichen vom Tastaturtreiber in einen Puffer abgelegt. Sobald ein Zeichen im Puffer steht, holt es die Hauptschleife und schickt es dem Focus-Rahmen (*HandleKey*-Meldung), der es behandelt, indem er es zum Beispiel an der Caret-Position einfügt. Anschließend gibt er die Kontrolle wieder zurück an die Hauptschleife.

Mausklicks

Solange keine Taste gedrückt wird, wird das Fenster, das den Mauszeiger enthält, von der Hauptschleife aufgefordert, auf die Maus zu reagieren (*HandleMouse*-Meldung). Normalerweise wird das Fenster nichts weiter tun als den Mauszeiger zu zeichnen. Wurde jedoch ein Mausknopf gedrückt, wird als Reaktion darauf zum Beispiel das Caret gesetzt, etwas selektiert oder gezeichnet. Anschließend geht die Kontrolle wieder an die Hauptschleife zurück.

Kooperatives Multitasking

Da Fenster in beliebiger Reihenfolge auf Ereignisse reagieren können und die Kontrolle schon nach kurzer Zeit wieder abgeben, entsteht der Eindruck, als ob alle Fenster gleichzeitig benutzbar wären, als ob also mehrere Programme gleichzeitig aktiv wären. In Wirklichkeit gibt

es jedoch nur einen einzigen Prozeß, der abwechselnd verschiedene Fenster bedient. Man nennt das *kooperatives Multitasking*.

Auf die gleiche Art werden auch im Oberon-System Benutzereingaben verarbeitet. Die Hauptschleife befindet sich dort im Modul Oberon, daher verpacken auch wir sie in ein Modul *Oberon0*, das eine sehr einfache Schnittstelle besitzt:

Schnittstelle von Oberon0

```
DEFINITION Oberon0;
PROCEDURE Loop;
END Oberon0.
```

Das Oberon0-System wird gestartet, indem man *Loop* (die Hauptschleife) aufruft. Es kann durch Drücken der Escape-Taste gestoppt werden. Der Quellcode von *Oberon0* sollte ohne Erklärungen verständlich sein. Die Module *Texts0* und *TextFrames0* werden im nächsten Kapitel erklärt. Das Modul *OS* ist in Anhang B beschrieben.

Implementierung von Oberon0

```
MODULE Oberon0;
IMPORT OS, Viewers0, Texts0, TextFrames0;

CONST
   ESC = 1BX;

PROCEDURE Loop*;
   VAR ch: CHAR; x, y: INTEGER; buttons: SET;
      v: Viewers0.Viewer; t: Texts0.Text;
BEGIN
   NEW(t); t.Clear;
   v := Viewers0.New( (*open log viewer*)
      TextFrames0.NewMenu("LOG", "Viewers0.Close"),     (*menu frame*)
      TextFrames0.New(t));                              (*text frame*)
   LOOP
      IF OS.AvailChars() > 0 THEN  (*if characters in input buffer*)
         OS.ReadKey(ch);
         IF ch = ESC THEN EXIT
         ELSIF Viewers0.focus # NIL THEN Viewers0.focus.HandleKey(ch)
         END
      ELSE
         OS.GetMouse(buttons, x, y);
         v := Viewers0.ViewerAt(y);
         IF v # NIL THEN v.HandleMouse(x, y, buttons)
         ELSE OS.DrawCursor(x, y)
         END
      END
   END
END Loop;

END Oberon0.
```

12.3 Ein Texteditor

Die wohl häufigste Art von Daten, die in Fenstern dargestellt werden, sind Texte. Wir wollen daher Klassen entwerfen und implementieren, die es erlauben, Text in einem Fenster anzuzeigen und zu editieren.

MVC-Schema

Welche Klassen und Module sind nötig? Erinnern wir uns an Kapitel 11.3, in dem wir als nützliche Technik zur Implementierung von Editoren das MVC-Schema eingeführt haben (Abb. 12.8).

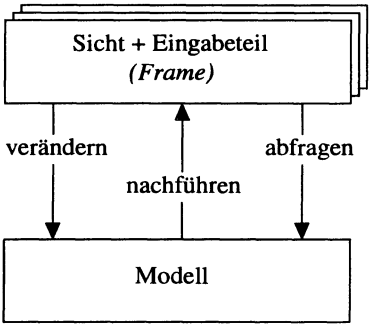


Abb. 12.8 Das MVC-Schema

Nach dem MVC-Schema besteht ein Editor aus einem *Datenmodell* sowie aus mehreren *Sichten* und *Eingabeteilen*. Wie ist das auf einen Texteditor zu übertragen?

Das Datenmodell ist der Text: wir brauchen also eine Klasse, die Text verwaltet. Die Funktion von Sicht und Eingabeteil wird von Rahmen wahrgenommen: wir brauchen also eine Klasse *TextFrame* – eine Unterklasse von *Frame* – die einen Text am Bildschirm anzeigt und Tastatureingaben und Mausklicks interpretiert. Der Textrahmen wird in einem Fenster angezeigt: dazu können wir die bereits existierende Klasse *Viewer* verwenden. Schließlich brauchen wir noch ein Modul, das Kommandos zum Öffnen und Schließen eines Textfensters enthält: wir nennen es *Edit0*. Jede der oben genannten Klassen implementieren wir in einem eigenen Modul; somit ergibt sich die in Abb. 12.9 gezeigte Modul- und Klassenhierarchie.

Fenster benutzen Textrahmen, indem sie ihnen Meldungen schicken und sie so auffordern, ihre Größe anzupassen oder ihren Inhalt anzuzeigen. *Viewers0* importiert aber *TextFrames0* nicht, sondern betrachtet alle Rahmen (auch Textrahmen) als Objekte der Basisklasse *Viewers0.Frame*. Ein Textrahmen wird mit *Viewers0.New* in einem Fenster installiert, ohne daß *Viewers0* diese Frame-Erweiterung kennt.

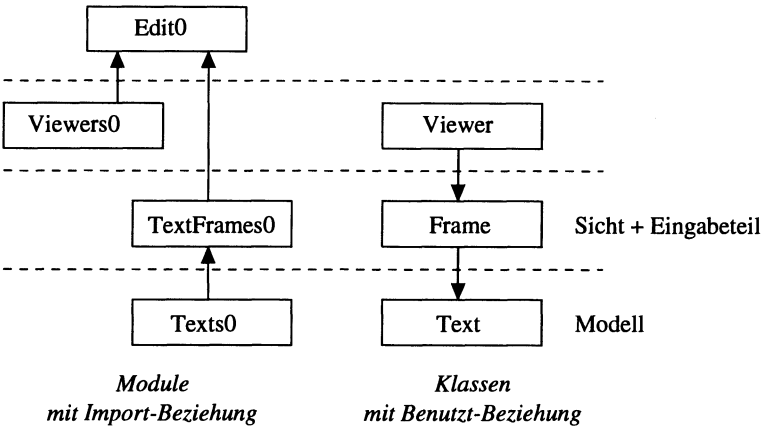


Abb. 12.9 Module und Klassen des Texteditors

Die Klasse *Text* hat eigentlich zwei Aufgaben: sie verwaltet ASCII-Text und verschiedene Schriftarten. Um diese beiden Aufgaben unabhängig voneinander beschreiben zu können, zerlegen wir sie in eine Klasse *AsciiText*, die den reinen Text verwaltet, und eine davon abgeleitete Klasse *Text*, die die Behandlung von Schriftarten hinzufügt.

Der Leser sei vorgewarnt: Ein Texteditor ist kein triviales Programm; seine Implementierung enthält zwangsläufig viele Details, obwohl hier versucht wurde, unnötige Einzelheiten unter Verzicht auf Effizienz und Allgemeinheit zu vermeiden.

12.3.1 Einfache Texte (AsciiTexts)

Der Klasse *AsciiTexts.Text* verwaltet einen Text als Folge von ASCII-Zeichen. Wünschenswerte Operationen auf Texten sind Einfügen, Löschen, zeichenweises Lesen und Schreiben, Laden und Speichern. Dies führt zu folgender Schnittstelle:

Schnittstelle von AsciiTexts

```
DEFINITION AsciiTexts;
IMPORT OS;

TYPE
   Text = POINTER TO TextDesc;
   TextDesc = RECORD (OS.ObjectDesc)
      len-: LONGINT;      (*text length*)
      pos-: LONGINT;      (*read/write position*)
      PROCEDURE (t: Text) Clear;
      PROCEDURE (t: Text) Insert (at: LONGINT;
         t1: Text; beg, end: LONGINT);
```

```
        PROCEDURE (t: Text) Delete (beg, end: LONGINT);
        PROCEDURE (t: Text) SetPos (pos: LONGINT);
        PROCEDURE (t: Text) Read (VAR ch: CHAR);
        PROCEDURE (t: Text) Write (ch: CHAR);
        PROCEDURE (t: Text) Load (VAR r: OS.Rider);
        PROCEDURE (t: Text) Store (VAR r: OS.Rider);
    END;
    NotifyInsMsg = RECORD (OS.Message) t: Text;  beg, end: LONGINT END;
    NotifyDelMsg = RECORD (OS.Message) t: Text;  beg, end: LONGINT END;

END AsciiTexts.
```

Text-Meldungen

Ein Text *t* ist eine Folge von *t.len* Zeichen mit den Positionen 0 bis *t.len*-1. Er besitzt eine Lese/Schreibposition *t.pos*, an der mit *Read* und *Write* gelesen und geschrieben werden kann. In den folgenden Erläuterungen bedeutet das offene Intervall [*a..b*[das Textstück beginnend mit dem Zeichen auf Position *a* und endend mit dem Zeichen auf Position *b*-1.

- *t.Clear* löscht den Inhalt des Texts *t*.
- *t.Insert(p, t1, a, b)* fügt das Textstück [*a..b*[aus *t1* an der Position *p* in *t* ein.
- *t.Delete(a, b)* löscht das Textstück [*a..b*[in *t*.
- *t.SetPos(p)* setzt die Lese/Schreibposition in *t* auf *p*.
- *t.Read(ch)* liest das Zeichen *ch* an der Lese/Schreibposition von *t* und erhöht sie um eins. Wenn versucht wird, über das Textende hinaus zu lesen, wird 0X geliefert und *pos* nicht erhöht.
- *t.Write(ch)* fügt das Zeichen *ch* an der Lese/Schreibposition von *t* ein und erhöht sie um eins.
- *t.Load(r)* lädt einen Text *t* von einer Datei (Rider *r*).
- *t.Store(r)* speichert den Text *t* auf eine Datei (Rider *r*).

Textpuffer

Die zentrale Datenstruktur eines Texts ist der *Textpuffer*. In seiner einfachsten Form ist er ein Array von Zeichen. Allerdings muß das Einfügen und Löschen von Zeichen effizient erfolgen, daher machen wir uns folgende Beobachtung zunutze:

Das Array ist normalerweise nicht zur Gänze gefüllt. Es besteht aus einer Folge von Zeichen und einer Lücke, die sich vom letzten Zeichen bis zum Ende des Arrays erstreckt. Das Einfügen und Löschen am Beginn dieser Lücke (d.h. am Textende) ist effizient, weil dabei keine Zeichen verschoben werden müssen. Inmitten des Textes ist es jedoch eine teure Operation.

Es hindert uns aber nichts daran, die Lücke vom Textende in das Innere des Textes zu verschieben. Dann kann man auch dort effizient einfügen und löschen (Abb. 12.10).

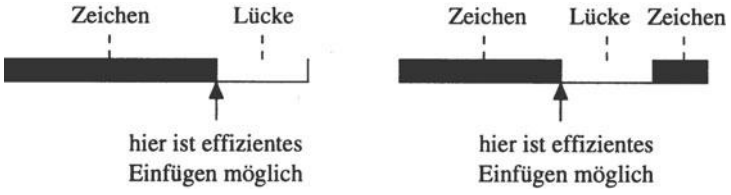


Abb. 12.10 Verschieben der Lücke vom Textende in das Innere des Textes

Jedesmal, wenn sich die Einfügeposition im Text ändert, wird die Lücke mittels einer Prozedur *MoveGap* an die neue Einfügeposition verschoben. Da man meist an einer Stelle gleich mehrere Zeichen eintippt, bevor man die Einfügeposition an eine andere Stelle setzt, muß die Lücke nur selten verschoben werden. Für die Klienten der Klasse *Text* bleibt natürlich das Array und die Position der Lücke verborgen. Man nennt diese Art von Texten *Lückentexte* (*gap texts*).

Arrays haben eine feste Länge. Um Texte beliebig lang werden zu lassen, ohne jedoch bei kleinen Texten zu viel Platz zu verschwenden, müssen wir erlauben, daß das Array wachsen und schrumpfen kann. Wir verwenden folgende Strategie: Wenn das Array bis zum letzten Zeichen gefüllt ist, legen wir ein größeres Array an und kopieren das alte in das neue um. Wird die Textlänge kürzer als die halbe Arraygröße, legen wir ein kleineres Array an und kopieren wieder um. Diese Aufgaben besorgen zwei Methoden *Grow* und *Shrink*.

Damit ist das Wesentliche über die Implementierung des Textpuffers gesagt, und wir können uns das Quellprogramm ansehen. Es sei noch vermerkt, daß Texte im Oberon-System anders implementiert sind als in Oberon0. Sie stehen nicht im Hauptspeicher, sondern auf einer Datei und können somit beliebig lang werden. Diese Implementierung ist praxisgerechter aber komplizierter als die in Oberon0 gewählte. Sie ist in [WiG92] beschrieben.

Implementierung von AsciiTexts

```
MODULE AsciiTexts;
IMPORT OS, Viewers0;

CONST minBufLen = 32;

TYPE
   Buffer = POINTER TO ARRAY OF CHAR;
   Text* = POINTER TO TextDesc;
   TextDesc* = RECORD (OS.ObjectDesc)
      len-: LONGINT;      (*text length*)
      pos-: LONGINT;      (*read/write position*)
      buf: Buffer;        (*text buffer*)
      gap: LONGINT        (*index of first byte in gap*)
   END;
```

```
    NotifyInsMsg* = RECORD(OS.Message)
        t*:Text; beg*, end*:LONGINT
    END;
    NotifyDelMsg* = RECORD(OS.Message)
        t*:Text; beg*, end*:LONGINT
    END;

PROCEDURE (t: Text) MoveGap (to: LONGINT);
    VAR n, gapLen: LONGINT;
BEGIN
    n := ABS(to - t.gap); gapLen := LEN(t.buf^) - t.len;
    IF to > t.gap THEN
        OS.Move(t.buf^, t.gap + gapLen, t.buf^, t.gap, n)
    ELSIF to < t.gap THEN
        OS.Move(t.buf^, t.gap - n, t.buf^, t.gap + gapLen - n, n)
    END;
    t.gap := to
END MoveGap;

PROCEDURE (t: Text) Grow (size: LONGINT);
    VAR bufLen: LONGINT; old: Buffer;
BEGIN
    bufLen := LEN(t.buf^);
    IF size > bufLen THEN
        t.MoveGap(t.len);
        WHILE bufLen < size DO bufLen := 2*bufLen END;
        old := t.buf; NEW(t.buf, bufLen); OS.Move(old^, 0, t.buf^, 0, t.len)
    END
END Grow;

PROCEDURE (t: Text) Shrink;
    VAR bufLen: LONGINT; old: Buffer;
BEGIN
    bufLen := LEN(t.buf^); t.MoveGap(t.len);
    WHILE (bufLen >= 2*t.len) & (bufLen > minBufLen) DO
        bufLen := bufLen DIV 2
    END;
    old := t.buf; NEW(t.buf, bufLen); OS.Move(old^, 0, t.buf^, 0, t.len)
END Shrink;

PROCEDURE (t: Text) Clear*;
BEGIN
    NEW(t.buf, minBufLen);
    t.gap := 0; t.pos := 0; t.len := 0
END Clear;

PROCEDURE (t: Text) Insert* (at: LONGINT; t1: Text; beg, end: LONGINT);
    VAR len: LONGINT; m: NotifyInsMsg; t0: Text;
BEGIN
    IF t = t1 THEN
        NEW(t0); t0.Clear; t0.Insert(0, t1, beg, end); t.Insert(at, t0, 0, t0.len)
    ELSE
        len := end - beg;
```

Text-Methoden

1 ☞

```
        IF t.len + len > LEN(t.buf^) THEN t.Grow(t.len + len) END;
        t.MoveGap(at); t1.MoveGap(end);
        OS.Move(t1.buf^, beg, t.buf^, t.gap, len);
        INC(t.gap, len); INC(t.len, len);
        m.t := t; m.beg := at; m.end := at + len; Viewers0.Broadcast(m)
    END
END Insert;

2 ☞ PROCEDURE (t: Text) Delete* (beg, end: LONGINT);
    VAR m: NotifyDelMsg;
BEGIN
    t.MoveGap(end); t.gap := beg; DEC(t.len, end-beg);
    IF (t.len * 2 < LEN(t.buf^)) & (LEN(t.buf^) > minBufLen) THEN t.Shrink END;
3 ☞ m.t := t; m.beg := beg; m.end := end; Viewers0.Broadcast(m)
END Delete;

PROCEDURE (t: Text) SetPos* (pos: LONGINT);
BEGIN
    t.pos := pos
END SetPos;

PROCEDURE (t: Text) Read* (VAR ch: CHAR);
    VAR i: LONGINT;
BEGIN
    i := t.pos;
    IF t.pos >= t.gap THEN INC(i, LEN(t.buf^) - t.len) END;
    IF t.pos < t.len THEN ch := t.buf[i]; INC(t.pos) ELSE ch := 0X END
END Read;

PROCEDURE (t: Text) Write* (ch: CHAR);
    VAR m: NotifyInsMsg;
BEGIN
    IF t.len = LEN(t.buf^) THEN t.Grow(t.len + 1) END;
    IF t.pos # t.gap THEN t.MoveGap(t.pos) END;
    t.buf[t.gap] := ch; INC(t.gap); INC(t.pos); INC(t.len);
    m.t := t; m.beg := t.gap-1; m.end := t.gap; Viewers0.Broadcast(m)
END Write;

PROCEDURE (t: Text) Load* (VAR r: OS.Rider);
    VAR len: LONGINT;
BEGIN
    t.Clear; r.ReadLInt(len); t.Grow(len);
    r.ReadChars(t.buf^, len);
    t.gap := len; t.len := len
END Load;

PROCEDURE (t: Text) Store* (VAR r: OS.Rider);
BEGIN
    t.MoveGap(t.len);
    r.WriteLInt(t.len); r.WriteChars(t.buf^, t.len)
END Store;

END AsciiTexts.
```

Die wichtigsten Methoden von *AsciiTexts* sind *Insert* und *Delete*. In *Insert* wird ein Textstück aus *t1* in einen Text *t* eingefügt, indem zuerst die Lücke in *t* an die Einfügestelle geschoben und dann das Textstück an diese Stelle kopiert wird (Abb. 12.11). Vorher muß *t* eventuell noch durch *t.Grow* auf die nötige Länge vergrößert werden. Wenn *t* und *t1* derselbe Text sind, wird ein Zwischenpuffer benutzt. 1 ☞

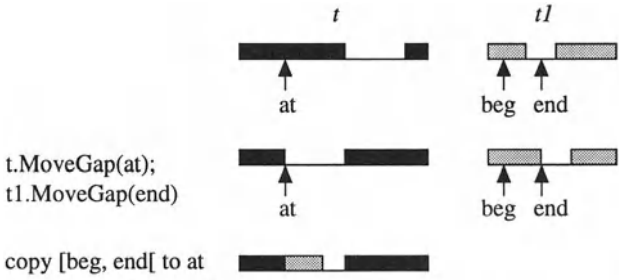


Abb. 12.11 Auswirkungen von *t.Insert(at, t1, beg, end)*

Delete funktioniert ähnlich. Die Lücke wird an das Ende des zu löschenden Textstücks verschoben und dann einfach nach vorne hin vergrößert (Abb. 12.12). Anschließend wird das Array in *t* wenn nötig mit *t.Shrink* verkleinert. Auch hier gilt: wenn mehrere Zeichen an derselben Stelle mittels der Tastatur gelöscht werden, muß die Lücke zwischen den Lösch-Operationen nicht verschoben werden. Das Löschen ist also in diesem häufigen Fall äußerst effizient. 2 ☞

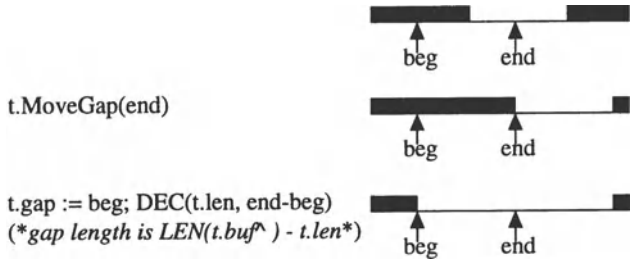


Abb. 12.12 Auswirkungen von *t.Delete(beg, end)*

Wenn sich der Text ändert, müssen seine Sichten davon informiert werden. Daher rufen *Insert*, *Delete* und *Write* die Prozedur *Viewers0.Broadcast* auf und übergeben ihr ein Meldungsobjekt, das angibt, auf welche Weise und an welcher Stelle der Text verändert wurde. *Broadcast* leitet die Meldung an alle Rahmen auf dem Bildschirm weiter. Diejenigen Rahmen, die den geänderten Text darstellen, reagieren, indem sie die Änderung am Bildschirm nachführen. 3 ☞

12.3.2 Texte mit Schriftarten und Elementen

Für einfache Texte mag *AsciiTexts.Text* bereits genügen. In einem realistischen Editor möchte man jedoch verschiedene Schriftarten (fonts) benutzen sowie Bilder und andere Elemente in den Text einbetten. Diese Eigenschaften implementieren wir in Form einer Klasse *Texts0.Text*, die eine Erweiterung von *AsciiTexts.Text* ist.

Schnittstelle von Texts0

```
DEFINITION Texts0;
IMPORT OS, AsciiTexts;

TYPE
   Style = POINTER TO StyleDesc;
   Element = POINTER TO ElemDesc;

   Text = POINTER TO TextDesc;
   TextDesc = RECORD (AsciiTexts.TextDesc)
      style-: Style;  (*style of previously read character*)
      PROCEDURE (t: Text) ChangeFont (beg, end: LONGINT; f: OS.Font);
      PROCEDURE (t: Text) ReadNextElem (VAR e: Element);
      PROCEDURE (t: Text) WriteElem (e: Element);
      PROCEDURE (t: Text) ElemPos (e: Element);
   END;

   StyleDesc = RECORD
      fnt-: OS.Font;      (*font of this style stretch*)
      elem-: Element   (*if not NIL, the corrsponding character is an element*)
   END;

   ElemDesc = RECORD (OS.ObjectDesc)
      w, h: INTEGER;  (*width and height of element in pixels*)
      dsc: INTEGER;  (*descender (part below the base line)*)
      PROCEDURE (e: Element) Draw (x, y: INTEGER);
      PROCEDURE (e: Element) HandleMouse
         (frame: OS.Object; x, y: INTEGER);
      PROCEDURE (e: Element) Copy (): Element;
   END;

   NotifyDelMsg = AsciiTexts.NotifyDelMsg;
   NotifyInsMsg = AsciiTexts.NotifyInsMsg;
   NotifyReplMsg= RECORD (OS.Message) t: Text; beg, end: LONGINT END;

END Texts0.
```

Text-Meldungen

Texts0.Text erbt die Schnittstelle von *AsciiTexts.Text*. Man kann also in Objekte dieser Klasse ebenfalls Textstücke einfügen, löschen usw. Allerdings werden die geerbten Methoden so überschrieben, daß Schriftarten und Elemente richtig mitgeführt werden. Zusätzlich werden folgende Operationen angeboten:

- *t.ChangeFont(a, b, fnt)* ändert die Schriftart des Textstücks [*a..b*[auf *fnt*.
- *t.ReadNextElem(e)* liefert das nächste Element *e* im Text *t* ab der Position *t.pos*. Anschließend enthält *t.pos* die Position des Zeichens nach *e*. Falls kein Element mehr gefunden wurde, ist *e* = NIL und *t.pos* = *t.len*.
- *t.WriteElem(e)* fügt das Element *e* an der Lese/Schreibposition in den Text *t* ein.
- *pos* := *t.ElemPos(e)* liefert die Position des Elements *e* im Text *t* oder den Wert *t.len* falls *e* nicht existiert.

Stile

Welche zusätzlichen Attribute werden in *Text* benötigt? Neben der eigentlichen Zeichenfolge braucht man auch Informationen über die Bilder und sonstigen Elemente im Text sowie über die Schriftarten. Diese Informationen nennen wir *Stile*. Sie werden in einer Stilliste verwaltet. Jeder Knoten dieser Liste ist vom Typ *Style* und steht für ein Textstück von *len* Zeichen, die alle die Schriftart *fnt* haben.

```
TYPE
   Style = POINTER TO StyleDesc;
   StyleDesc = RECORD
      len: LONGINT;   (*length of this style stretch*)
      fnt: OS.Font;   (*font of this style stretch*)
      elem: Element;  (*pointer to element descriptor or NIL*)
      next: Style
   END;
```

Elemente werden im Text durch das Zeichen 1CX und in der Stilliste durch einen Knoten mit *len* = 1 dargestellt; *elem* enthält in diesen Knoten das eigentliche Element. Bei gewöhnlichen Zeichen hat *elem* den Wert NIL. Bevor wir auf die Eigenschaften und das Verhalten von Elementen eingehen, kümmern wir uns um die Verwaltung der Stilliste. Abb. 12.13 zeigt den Zusammenhang zwischen ASCII-Text und Stilliste. Die Stilliste ist für Klienten von *Text* nicht sichtbar.

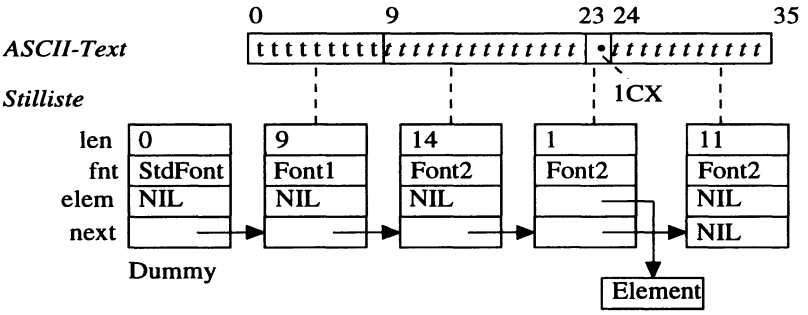


Abb. 12.13 Ein Text mit zugehöriger Stilliste

Der erste Knoten der Stilliste ist ein Hilfsknoten, der die Verwaltung der Liste erleichtert. Jeder Knoten enthält die Länge des Textstücks, das er beschreibt, aber nicht dessen Position, da sonst nach jedem eingefügten Zeichen die dahinter liegenden Positionen nachgeführt werden müßten.

Elemente

Ein Text soll auch Objekte enthalten können, die keine Zeichen sind, sondern Bilder, Tabellen oder Formeln (siehe Kapitel 11.4). Es ist nicht voraussehbar, welche Arten solcher Objekte es einmal geben wird. Wir wollen Texte auch nicht dadurch aufblähen, daß sie unnötig viele Objektarten kennen müssen. Daher nehmen wir an, daß Texte nicht zwischen den Objektarten unterscheiden, sondern mit einer abstrakten Klasse *Element* arbeiten, aus der später Grafikelemente oder Tabellenelemente abgeleitet werden können. Das hält den Editor klein und gibt uns die Möglichkeit, später jederzeit neue Elementarten hinzuzufügen.

```
TYPE
   Element = POINTER TO ElemDesc;
   ElemDesc = RECORD (OS.ObjectDesc)
      w, h: INTEGER; (*width and height of element in pixels*)
      dsc: INTEGER   (*descender (part below the base line)*)
   END;
```

Welche Operationen sollen auf Elemente anwendbar sein? Elemente sollen sich am Bildschirm darstellen können und auf Mausklicks reagieren. Man muß sie auf eine Datei schreiben und einlesen können, was allerdings bereits eine Eigenschaft von Objekten der Basisklasse *OS.Object* ist. Elemente müssen also folgende Meldungen verstehen:

Element-Meldungen

- *e.Draw(x, y)* zeichnet das Element *e* an der Position *(x, y)* am Bildschirm (Abb. 12.14).
- *e.HandleMouse(f, x, y)* reagiert auf einen Mausklick an der Position *(x, y)* im Rahmen *f*.
- *e1 := e.Copy()* liefert eine Kopie von *e*.

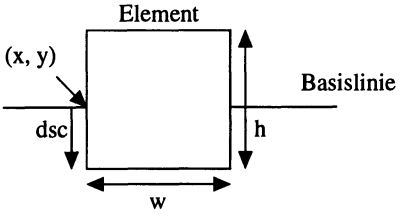


Abb. 12.14 Koordinaten eines Elements am Bildschirm

Die meisten Methoden der Basisklasse *Element* sind leer und werden normalerweise in den Unterklassen überschrieben.

Implementierung von Texts0

```
MODULE Texts0;
IMPORT OS, AsciiTexts, Viewers0;

CONST ELEM = 1CX;

TYPE
   Element* = POINTER TO ElemDesc;
   Style* = POINTER TO StyleDesc;

   Text* = POINTER TO TextDesc;
   TextDesc* = RECORD (AsciiTexts.TextDesc)
      style-: Style;          (*style of previously read character*)
      firstStyle: Style;      (*to style list (first node is dummy)*)
      styleRest: LONGINT   (*unread bytes in current style stretch*)
   END;

   StyleDesc* = RECORD
      len: LONGINT;   (*length of style stretch*)
      fnt-: OS.Font;  (*font of this style stretch*)
      elem-: Element; (*pointer to element descriptor or NIL*)
      next: Style
   END;

   ElemDesc* = RECORD (OS.ObjectDesc)
      w*, h*: INTEGER;  (*width and height in pixels*)
      dsc*: INTEGER     (*descender (part under the base line)*)
   END;

   NotifyInsMsg* = AsciiTexts.NotifyInsMsg;
   NotifyDelMsg* = AsciiTexts.NotifyDelMsg;
   NotifyReplMsg* = RECORD (OS.Message)
      t*: Text; beg*, end*: LONGINT
   END;
```

Element-Methoden

```
PROCEDURE (e: Element) Draw* (x, y: INTEGER);
END Draw;

PROCEDURE (e: Element) HandleMouse* (f: OS.Object; x, y: INTEGER);
END HandleMouse;

PROCEDURE (e: Element) Copy* (): Element;
END Copy;

PROCEDURE (e: Element) Load* (VAR r: OS.Rider);
BEGIN
   r.ReadInt(e.w); r.ReadInt(e.h); r.ReadInt(e.dsc)
END Load;

PROCEDURE (e: Element) Store* (VAR r: OS.Rider);
BEGIN
```

```
    r.WriteInt(e.w); r.WriteInt(e.h); r.WriteInt(e.dsc)
END Store;
```

Text-Methoden

1 ☞

```
PROCEDURE (t: Text) Split (pos: LONGINT; VAR prev: Style);
    VAR a, b: Style;
BEGIN
    a := t.firstStyle;
    WHILE (a # NIL) & (pos >= a.len) DO
        DEC(pos, a.len); prev := a; a := a.next
    END;
    IF (a # NIL) & (pos > 0) THEN
        NEW(b); b.elem := a.elem; b.fnt := a.fnt; b.len := a.len - pos; a.len := pos;
        b.next := a.next; a.next := b; prev := a
    END
END Split;
```

2 ☞

```
PROCEDURE (t: Text) Merge (a: Style);
    VAR b: Style;
BEGIN
    b := a.next;
    IF (b#NIL) & (a.fnt=b.fnt) & (a.len>0) & (a.elem=NIL) & (b.elem=NIL) THEN
        INC(a.len, b.len); a.next := b.next
    END
END Merge;
```

3 ☞

```
PROCEDURE (t: Text) Insert*
    (at: LONGINT; t1: AsciiTexts.Text; beg, end: LONGINT);
    VAR a, b, c, d, i, j, k: Style; t0: Text;
BEGIN
    IF t = t1 THEN
        NEW(t0); t0.Clear; t0.Insert(0, t1, beg, end); t.Insert(at, t0, 0, t0.len)
    ELSE
        WITH t1: Text DO
            t1.Split(beg, a); t1.Split(end, b); t.Split(at, c); d := c.next;
            i := a; j := c;
            WHILE i # b DO
                i := i.next; NEW(k); k^ := i^;
                IF i.elem # NIL THEN k.elem := i.elem.Copy() END;
                j.next := k; j := k
            END;
            j.next := d; t1.Merge(b); t1.Merge(a); t.Merge(j); t.Merge(c);
            t.Insert^ (at, t1, beg, end)
        END
    END
END Insert;

PROCEDURE (t: Text) Delete* (beg, end: LONGINT);
    VAR a, b: Style;
BEGIN
    t.Split(beg, a); t.Split(end, b); a.next := b.next; t.Merge(a);
    t.Delete^ (beg, end)
END Delete;
```

```
PROCEDURE (t: Text) SetPos* (pos: LONGINT);
    VAR prev, a: Style;
BEGIN
    t.SetPos^(pos);
    a := t.firstStyle;
    WHILE (a # NIL) & (pos >= a.len) DO
        DEC(pos, a.len); prev := a; a := a.next
    END;
    IF (a = NIL) OR (pos = 0) THEN t.style := prev; t.styleRest := 0
    ELSE t.style := a; t.styleRest := a.len-pos
    END
END SetPos;

PROCEDURE (t: Text) Read* (VAR ch: CHAR);                                  4 ☞
BEGIN
    t.Read^(ch);
    IF (t.styleRest = 0) & (t.style.next # NIL) THEN
        t.style := t.style.next; t.styleRest := t.style.len
    END;
    DEC(t.styleRest)
END Read;

PROCEDURE (t: Text) Write* (ch: CHAR);                                     5 ☞
    VAR a, prev: Style; at: LONGINT;
BEGIN
    a := t.firstStyle; at := t.pos;
    WHILE (a # NIL) & (at >= a.len) DO DEC(at, a.len); prev := a; a := a.next
END;
    IF (a = NIL) OR (at = 0) THEN  (*insert at end of style stretch*)
        IF (prev = t.firstStyle) OR (prev.elem # NIL) THEN
            NEW(a); a.elem := NIL; a.fnt := prev.fnt; a.len := 1;
            a.next := prev.next; prev.next := a;
            t.Merge(a)
        ELSE INC(prev.len)
        END
    ELSE INC(a.len)
    END;
    t.Write^ (ch)
END Write;

PROCEDURE (t: Text) ReadNextElem* (VAR e: Element);
    VAR pos: LONGINT; a: Style;
BEGIN
    pos := t.pos + t.styleRest;
    a := t.style.next;
    WHILE (a # NIL) & (a.elem = NIL) DO pos := pos + a.len; a := a.next END;
    IF a # NIL THEN e := a.elem; t.SetPos(pos+1)
    ELSE e := NIL; t.SetPos(t.len)
    END
END ReadNextElem;

PROCEDURE (t: Text) WriteElem* (e: Element);                               6 ☞
    VAR x, y: Style; m: NotifyReplMsg;
```

```
BEGIN
    t.Write(ELEM); t.Split(t.pos - 1, x); t.Split(t.pos, y); y.elem := e;
    m.t := t; m.beg := t.pos-1; m.end := t.pos; Viewers0.Broadcast(m)
END WriteElem;

PROCEDURE (t: Text) ElemPos* (e: Element): LONGINT;
    VAR pos: LONGINT; a: Style;
BEGIN
    a := t.firstStyle; pos := 0;
    WHILE (a # NIL) & (a.elem # e) DO pos := pos + a.len; a := a.next END;
    RETURN pos
END ElemPos;

PROCEDURE (t: Text) ChangeFont* (beg, end: LONGINT; fnt: OS.Font);
    VAR a, b: Style; m: NotifyReplMsg;

    PROCEDURE Change(a: Style);
    BEGIN
        a.fnt := fnt;
        IF a # b THEN Change(a.next) END;
        t.Merge(a)
    END Change;

BEGIN
    IF end > beg THEN
        t.Split(beg, a); t.Split(end, b); Change(a.next); t.Merge(a);
        m.t := t; m.beg := beg; m.end := end; Viewers0.Broadcast(m)
    END
END ChangeFont;

PROCEDURE (t: Text) Clear*;
BEGIN
    t.Clear^;
    NEW(t.firstStyle); t.firstStyle.elem := NIL; t.firstStyle.next := NIL;
    t.firstStyle.fnt := OS.DefaultFont(); t.firstStyle.len := 0; t.SetPos(0)
END Clear;
```

7 ☞

```
PROCEDURE (t: Text) Store* (VAR r: OS.Rider);
    VAR a: Style;
BEGIN
    t.Store^(r); a := t.firstStyle.next;
    WHILE a # NIL DO
        r.WriteString(a.fnt.name);
        r.WriteObj(a.elem); r.WriteLInt(a.len);
        a := a.next
    END;
    r.Write(0X)  (*empty font name terminates style list*)
END Store;

PROCEDURE (t: Text) Load* (VAR r: OS.Rider);
    VAR prev, a: Style; name: ARRAY 32 OF CHAR; x: OS.Object;
BEGIN
    t.Load^(r);
```

```
    prev := t.firstStyle;
    LOOP
        r.ReadString(name); IF name = "" THEN EXIT END;
        NEW(a); a.fnt := OS.FontWithName(name);
        r.ReadObj(x); r.ReadLInt(a.len);
        IF x = NIL THEN a.elem := NIL ELSE a.elem := x(Element) END;
        prev.next := a; prev := a
    END;
    prev.next := NIL
END Load;

END Texts0.
```

Die Stilliste wird durch zwei Grundoperationen *Split* und *Merge* verwaltet. *t.Split(pos, a)* spaltet ein Stilstück an der Position *pos* und erzeugt daraus zwei (Abb. 12.15). Das Stilstück *a* vor der Spaltungsposition wird zurückgegeben. 1 ☞

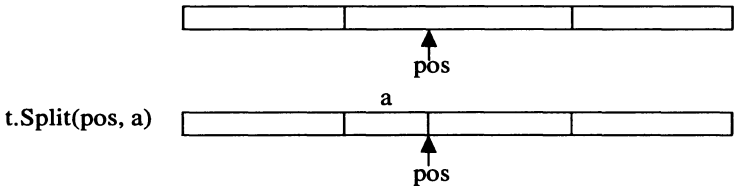


Abb. 12.15 Auswirkungen von *t.Split(pos, a)* auf die Stilliste

Die inverse Operation *t.Merge(a)* verschmilzt das Stilstück *a* mit seinem Nachfolger, falls beide die gleiche Schriftart haben und keine Elemente darstellen. 2 ☞

Die komplizierteste Operation von *Texts0.Text* ist *Insert*. Sie fügt ein Stück eines Textes *t1* in einen anderen Text *t* ein. Dabei müssen Stilstücke vorübergehend gespalten und nachher wieder verschmolzen werden, wie das in Abb. 12.16 gezeigt wird. Wenn *t* und *t1* derselbe Text sind, wird mit einem Zwischenpuffer gearbeitet. 3 ☞
Der Einfachheit halber nehmen wir an, daß *t1* und *t* vom gleichen dynamischen Typ *Texts0.Text* sind. Da Parametertypen beim Überschreiben von Methoden nicht geändert werden dürfen, muß *t1* mit dem statischen Typ *AsciiTexts.Text* deklariert werden. Es ist daher eine With-Anweisung (Typzusicherung) nötig, um *t1* als *Texts0.Text*-Objekt behandeln zu können.

Beim Lesen eines Zeichens mit *t.Read(ch)* wird sein Stil im Feld *t.style* abgelegt. Solange die gelesenen Zeichen zum gleichen Stilstück gehören, ist es nicht nötig *t.style* jedesmal neu zu setzen. Daher wird die Anzahl der im laufenden Stilstück noch ungelesenen Zeichen im Feld *t.styleRest* gespeichert. Wenn *t.styleRest* 0 ist, muß *t.style* auf das nächste Stilstück gesetzt werden. 4 ☞

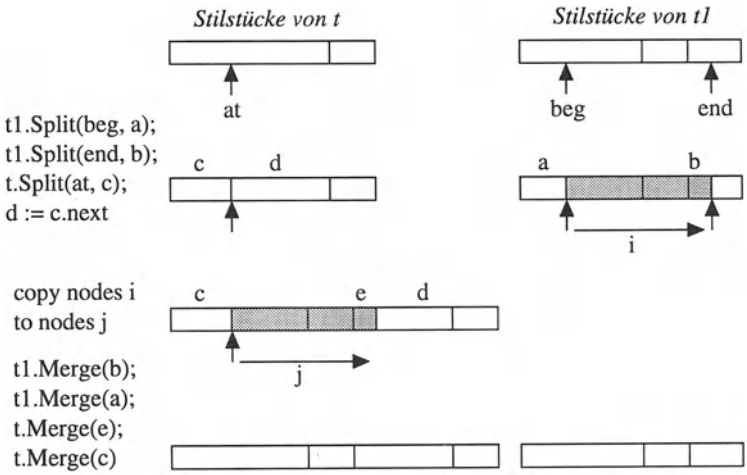


Abb. 12.16 Auswirkungen von *t.Insert(at, t1, beg, end)* auf die Stilliste

5 ☞ Beim Schreiben eines Zeichens mit *t.Write(ch)* kann man keine Stile setzen. Man kann aber die Schriftart eines geschriebenen Textstücks mit *t.ChangeFont* ändern.

Zeichen, die an einer Stilstückgrenze eingegeben werden, gehören zum davorliegenden Stilstück. Fügt man sie jedoch unmittelbar hinter einem Element ein, wird ein neues Stilstück angelegt (Stilstücke von Elementen müssen immer die Länge 1 haben).

6 ☞ Elemente werden mit einer besonderen Prozedur *WriteElem* eingefügt. Dabei wird ein neues Stilstück für sie angelegt.

7 ☞ Wenn ein Text auf eine Datei gespeichert wird, müssen seine Stile mitgespeichert werden. Für Schriftarten wird ihr Name ausgegeben; Elemente werden mittels der Methode *WriteObj* abgespeichert, die – wie in Kapitel 9.4.6 beschrieben – Typ und Wert des Elements ausgibt. Elemente werden dabei aufgefordert, sich selbst abzuspeichern, da nur sie ihre Struktur kennen. Den Abschluß der Stilliste auf der Datei bildet ein leerer Schriftartenname.

Klassen als Strukturierungsmittel

Was kann man aus dieser Implementierung lernen? In *AsciiTexts* und *Texts0* wurden Klassen als Strukturierungsmittel eingesetzt. Sie gliedern die Textverwaltung in zwei unabhängige Aufgabenbereiche: die Textpufferverwaltung und die Stilverwaltung. Nach dem Grundsatz, daß ein Baustein möglichst nur eine einzige Aufgabe wahrnehmen soll, wurden sie verschiedenen Klassen zugewiesen: die Textpufferverwaltung *AsciiTexts* und die Stilverwaltung *Texts0*.

Klassen als Halbfabrikate

AsciiTexts.Text ist für sich bereits ein nützlicher Baustein. Für einfache Texte, bei denen man keine verschiedenen Schriftarten braucht,

reicht er vollkommen. Gleichzeitig stellt er ein Halbfabrikat dar, das bei Bedarf zu einem Endfabrikat ausgebaut werden kann.

Erweiterbarkeit

Wir haben Vorsorge getroffen, daß Texte erweiterbar sind. Die abstrakte Klasse *Element* dient als "Steckplatz", in den beliebige *Element*-Erweiterungen eingesteckt und zusammen mit *Text* benutzt werden können. Ein Texteditor mit erweiterbaren Elementen ist auch im Oberon-System vorhanden [Szy92] und hat sich als äußerst nützlich und vielseitig erwiesen. Ein Beispiel für eine *Element*-Erweiterung wird in Kapitel 12.5 behandelt.

12.3.3 Editieren von Text (TextFrames0)

Wir können nun Texte mit verschiedenen Schriftarten verwalten, aber sie weder am Bildschirm anzeigen noch editieren. Was fehlt, sind die aus dem MVC-Schema bekannten Komponenten für die Datensicht und die Eingabebehandlung des Editors. Gemäß Abb. 12.8 fassen wir sie zu einer Klasse *Frame* zusammen, die aus *Viewers0.Frame* abgeleitet ist und einen Textrahmen darstellt.

Aufgaben von Textrahmen

Ein Textrahmen ist ein rechteckiger Bildschirmbereich, der in ein Fenster eingebettet wird und folgende Aufgaben hat:

1. *Text darstellen.* Der Text als kontinuierlicher Zeichenstrom wird in Zeilen "gegossen" und angezeigt. Dabei wird jedes Zeichen durch eine rechteckige Punktematrix dargestellt. Die Zeichen werden gemäß ihrer Breite hintereinander in eine Zeile gestellt; beim Auftreten eines Zeilenende-Symbols wird eine neue Zeile begonnen (Abb. 12.17). Am linken Rand des Textrahmens befindet sich ein sogenannter Rollbalken zum Blättern im Text.

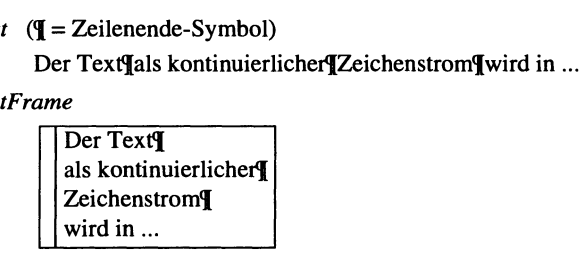
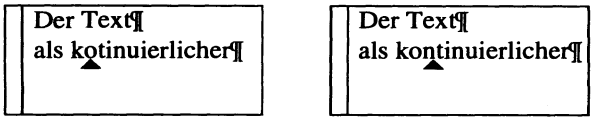


Abb. 12.17 Gießen eines Textes in einen Textrahmen

2. *Tastatureingaben verarbeiten.* Das eingetippte Zeichen wird an der Caret-Position eingefügt und der Rest der Zeile nach rechts verschoben (Abb. 12.18).



Nach Eingabe des Zeichens n

Abb. 12.18 Verarbeitung von Tastatureingaben

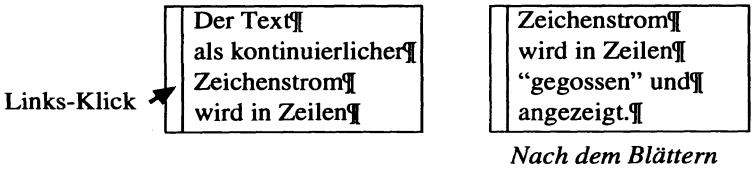
3. *Mausklicks verarbeiten.* Wird einer der drei Mausknöpfe gedrückt, so reagiert der Rahmen gemäß Tabelle 12.1 darauf, je nachdem, ob der Mausknopf im Textbereich oder im Rollbalken des Rahmens gedrückt wurde.

Tabelle 12.1 Reaktion eines Textrahmens auf Mausklicks

Mausknopf	im Textbereich	im Rollbalken
links	Caret setzen	vorwärts blättern
mittel	Kommando ausführen	absolut positionieren
rechts	selektieren	zum Textanfang blättern
rechts + links	Selektion löschen	
rechts + mittel	Selektion zum Caret kopieren	

Wird der mittlere Mausknopf im Textbereich geklickt, so wird das getroffene Wort als Kommando (der Form *Modul.Prozedur*) interpretiert und ausgeführt. Bei einem Rechts-Klick werden alle Zeichen, die bis zum Loslassen des Mausknopfs überstrichen werden, als selektiert betrachtet und durch invertierte Darstellung angezeigt.

Beim *Vorwärtsblättern* durch einen Links-Klick in den Rollbalken wird die Zeile auf der Höhe des Mausklicks zur ersten Zeile im Rahmen (Abb. 12.19). Beim *absoluten Positionieren* durch einen Mittel-Klick in den Rollbalken bestimmt die vertikale Position der Maus im Rollbalken, ab welcher Textposition der Text im Rahmen angezeigt werden soll. Ein Klick auf mittlerer Höhe des Rollbalkens bewirkt also, daß die Mitte des Textes ganz oben im Rahmen erscheint. Ein Klick im unteren Ende des Rollbalkens läßt das Ende des Textes oben im Rahmen erscheinen.



Nach dem Blättern

Abb. 12.19 Vorwärtsblättern durch Links-Klick in den Rollbalken

Aus Einfachheitsgründen unterstützen unsere Textrahmen kein Rückwärtsblättern. Man kann allerdings durch einen Rechts-Klick in den Rollbalken ganz an den Anfang des Textes gelangen.

Wir implementieren Textrahmen als eine Klasse *Frame* in einem Modul namens *TextFrames0* mit folgender Schnittstelle:

Schnittstelle von TextFrames0

```
DEFINITION TextFrames0;
IMPORT OS, Viewers0, Texts0;

TYPE
   Position = RECORD     (*position of a character c on the screen*)
      x-, y-: INTEGER;   (* left point on base line*)
      dx-: INTEGER;      (*width of c*)
      org-: LONGINT;     (*origin of line containing c*)
      pos-: LONGINT      (*text position of c*)
   END;

   Frame = POINTER TO FrameDesc;
   FrameDesc = RECORD (Viewers0.FrameDesc)
      text: Texts0.Text;     (*text displayed in this frame*)
      org-: LONGINT;         (*origin: text pos. of first char. in frame*)
      caret-: Position;      (*caret.pos < 0: no caret visible*)
      selBeg-, selEnd-: Position;     (*selBeg.pos < 0: no selection visible*)
      PROCEDURE (f: Frame) Draw;
      PROCEDURE (f: Frame) Defocus;
      PROCEDURE (f: Frame) Neutralize;
      PROCEDURE (f: Frame) Modify (dy: INTEGER);
      PROCEDURE (f: Frame) HandleKey (ch: CHAR);
      PROCEDURE (f: Frame) HandleMouse (x, y: INTEGER; but: SET);
      PROCEDURE (f: Frame) Handle (VAR m: OS.Message);
      PROCEDURE (f: Frame) SetCaret (pos: LONGINT);
      PROCEDURE (f: Frame) RemoveCaret;
      PROCEDURE (f: Frame) SetSelection (from, to: LONGINT);
      PROCEDURE (f: Frame) RemoveSelection;
      PROCEDURE (f: Frame) Copy (): Viewers0.Frame;
   END;

VAR
   cmdFrame-: Frame;     (*frame containing the most recent command*)
   cmdPos-: LONGINT;     (*text position after the most recent command*)

PROCEDURE New (t: Texts0.Text): Frame;
PROCEDURE NewMenu (name, commands: ARRAY OF CHAR): Frame;
PROCEDURE GetSelection (VAR f: Frame);

END TextFrames0.
```

Position

Der Typ *Position* beschreibt den Ort eines Zeichens *ch* am Bildschirm. Er dient zur Speicherung der Position des Carets und der Selektion. Die Felder *x*, *y* und *dx* bezeichnen die Bildschirmkoordina-

ten und Ausdehnung von *ch* in Pixel (Abb. 12.20 und Abb. 12.22). Das Feld *pos* gibt die Textposition von *ch* an und *org* die Textposition des ersten Zeichens der Zeile, in der *ch* steht.

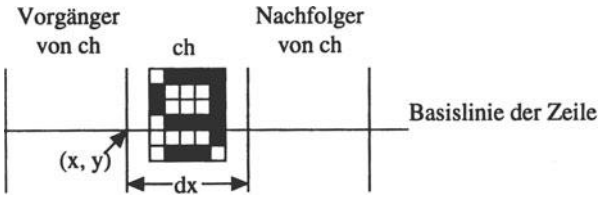


Abb. 12.20 Bedeutung der Felder *x*, *y* und *dx* des Typs *Position*

Frame-Meldungen

Der wichtigste Typ in *TextFrames0* ist die Klasse *Frame*. Ihre Schnittstelle wird zum Teil von *Viewers0.Frame* geerbt, zum Teil kommen aber auch textspezifische Meldungen hinzu.

- *f.Draw* gießt den ganzen Text von *f* neu.
- *f.Defocus* entfernt das Caret durch Aufruf von *f.RemoveCaret*
- *f.Neutralize* entfernt die Selektion und das Caret durch Aufrufe von *f.RemoveSelection* und *f.RemoveCaret*.
- *f.Modify(dy)* verschiebt den unteren Rand von *f* um *dy* und gießt eventuell neu sichtbar gewordenen Text.
- *f.HandleKey(ch)* fügt *ch* an der Caret-Position ein.
- *f.HandleMouse(x, y, b)* reagiert auf einen Mausklick an der Position *(x, y)* relativ zur linken unteren Ecke des Bildschirms. *b* gibt die Menge der gedrückten Mausknöpfe an.
- *f.Handle(m)* reagiert auf Notify-Meldungen *m*, die der Text verschickt, wenn er verändert wird (siehe Kapitel 11.3)
- *f.SetCaret(pos)* setzt das Caret auf die Position *pos*.
- *f.RemoveCaret* entfernt das Caret.
- *f.SetSelection(a, b)* setzt die Selektion im Intervall [*a*..*b*[.
- *f.RemoveSelection* entfernt die Selektion.
- *f1 := f.Copy()* liefert eine Kopie von *f*.

Die Prozedur *New* liefert einen neuen Textrahmen. *NewMenu* liefert einen neuen Menürahmen, das heißt einen Textrahmen, der einen Fensternamen und Menükommandos enthält. *GetSelection* sucht in allen am Bildschirm sichtbaren Textrahmen nach der jüngsten Selektion und liefert den Rahmen, der sie enthält oder NIL, wenn keine Selektion sichtbar ist. Dazu ist es notwendig, daß jeder Textrahmen zu seiner Selektion eine (nicht exportierte) *Zeitmarke* speichert, die angibt, wann die Selektion gesetzt wurde.

Bevor wir uns der Implementierung von *TextFrames0* zuwenden, die naturgemäß ziemlich kompliziert ist, wollen wir noch einige der verwendeten Datenstrukturen näher betrachten.

Rahmen-Metrik

Der Bereich eines Textrahmens wird in einen *Textbereich* und einen *Rollbalken* unterteilt. Der Textbereich besitzt einen Rand *margin*, in dem kein Text dargestellt wird (Abb. 12.21).

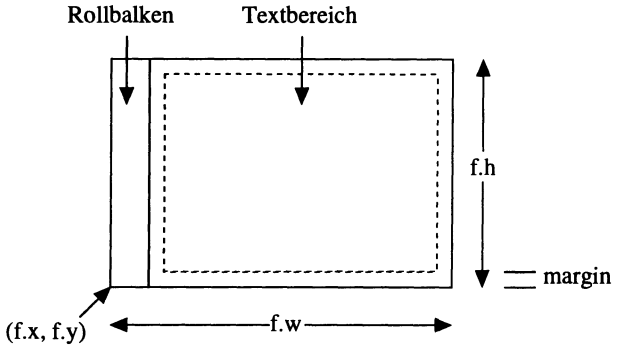


Abb. 12.21 Rahmen-Metrik

Zeichen-Metrik

Jedem Zeichen entspricht ein Punktemuster *pat*, das gemäß Abb. 12.22 in einem Rechteck der Breite *dx* und der Höhe *asc+dsc* am Bildschirm dargestellt wird. Die Zeichenmetrik (*x*, *y*, *w*, *h*, *dx*, *asc*, *dsc*) wird der entsprechenden Schriftart entnommen. Bei einem Element *e* (z.B. einem Bild) ist das Rechteck durch seine Breite *e.w*, seine Höhe *e.h* und seinen Abstand *e.dsc* zur Basislinie bestimmt (siehe Abb. 12.14).

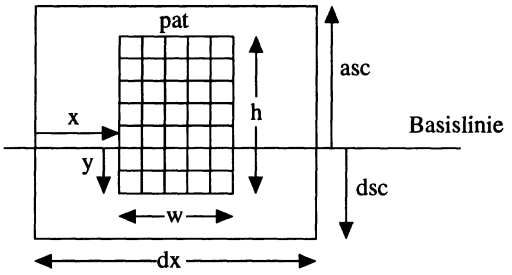


Abb. 12.22 Zeichen-Metrik

Zeilen-Metrik

Die Rechtecke aufeinanderfolgender Zeichen werden aneinandergereiht und bilden eine Zeile. Bevor eine Zeile am Bildschirm angezeigt wird, muß sie vermessen werden. Dazu wird ihre Länge in Zeichen (*len*) und Pixeln (*wid*) bestimmt sowie ihre Höhe (*asc+dsc*), die sich

aus den Höhen der einzelnen Zeichen oder Elemente ergibt (Abb. 12.23).

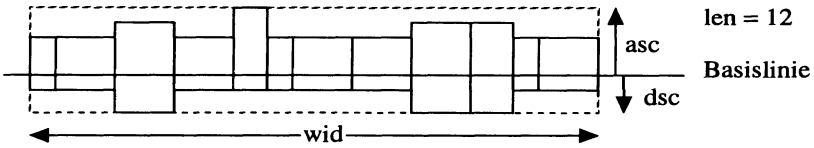


Abb. 12.23 Zeilen-Metrik

Die Metrik jeder Zeile wird in einem *Zeilendeskriptor* der folgenden Form gespeichert:

```
TYPE
   Line = POINTER TO LineDesc;
   LineDesc = RECORD
      len, wid: INTEGER;          (*length, width*)
      asc, dsc: INTEGER;          (*ascender, descender*)
      eol: BOOLEAN;               (*TRUE if line is terminated with EOL*)
      next: Line
   END;
```

Die Deskriptoren der am Bildschirm sichtbaren Zeilen sind in einer Ringliste miteinander verkettet (Abb. 12.24). Man beachte, daß ein Zeilendeskriptor nicht den *Text* der Zeile enthält, sondern nur ihre Maße. Der Text wird bei Bedarf jedesmal von der Datei gelesen. Das ermöglicht beliebig lange Zeilen bei geringem Speicheraufwand.

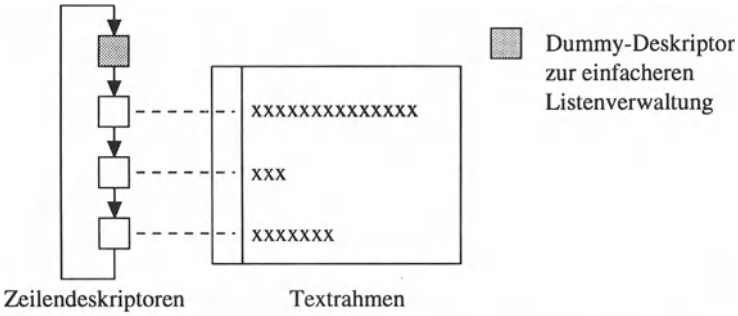


Abb. 12.24 Liste von Zeilendeskriptoren für einen Textrahmen

Der Leser sollte nun in der Lage sein, den Quelltext von *TextFrames0* zu verstehen. Er möge jedoch Papier und Bleistift bereithalten, denn ohne Hilfsskizzen dürfte manches schwierig nachvollziehbar sein. *TextFrames0* ist bei weitem das komplexeste Modul dieser Fallstudie.

Implementierung von TextFrames0

```
MODULE TextFrames0;
IMPORT OS, Viewers0, Texts0;

CONST
    EOL = 0DX; DEL = 7FX;      (*end of line character; delete character*)
    scrollW = 12;              (*width of scroll bar*)

TYPE
    Line = POINTER TO LineDesc;
    LineDesc = RECORD
        len, wid, asc, dsc: INTEGER;  (*length, width, ascender, descender*)
        eol: BOOLEAN;                 (*TRUE if line is terminated with EOL*)
        next: Line
    END;
    Position* = RECORD        (*position of a character c on the screen*)
        x-, y-, dx-: INTEGER;  (*(x,y) = left point on base line; dx = width of c*)
        org-, pos-: LONGINT;   (*origin of line containing c; text position of c*)
        L: Line                (*line containing c*)
    END;
    Frame* = POINTER TO FrameDesc;
    FrameDesc* = RECORD (Viewers0.FrameDesc)
        text*: Texts0.Text;
        org-: LONGINT;            (*index of first character in the frame*)
        caret-: Position;         (*caret; visible if caret.pos >= 0*)
        selBeg-, selEnd-: Position; (*selection; visible if selBeg.pos >= 0*)
        selTime: LONGINT;         (*time stamp of selection*)
        lsp: INTEGER;             (*space between lines*)
        margin: INTEGER;          (*space between frame border and text*)
        lines: Line               (*list of lines in frame (first line in dummy)*)
    END;
    SelectionMsg = RECORD (OS.Message) f: Frame END;

VAR
    cmdFrame-: Frame;     (*frame containing the most recent command*)
    cmdPos-: LONGINT;     (*text position after the most recent command*)
```

Hilfsprozeduren

```
PROCEDURE GetMetric (at: Texts0.Style; ch: CHAR;
        VAR dx, x, y, asc, dsc: INTEGER; VAR pat: OS.Pattern);
    VAR w, h: INTEGER;
BEGIN
    IF at.elem = NIL THEN
        OS.GetCharMetric(at.fnt, ch, dx, x, y, w, h, pat);
        asc := at.fnt.maxY; dsc := - at.fnt.minY
    ELSE
        dx := at.elem.w; x := 0; y := 0; dsc := at.elem.dsc; asc := at.elem.h - dsc
    END
END GetMetric;

PROCEDURE MeasureLine (t: Texts0.Text; VAR L: Line);
    VAR ch: CHAR; dx, x, y, asc, dsc: INTEGER; pat: OS.Pattern;
BEGIN
    L.len := 0; L.wid := 0; L.asc := 0; L.dsc := 0; ch := " ";
    WHILE (ch # EOL) & (t.pos < t.len) DO
```

1 ☞

```
            t.Read(ch); INC(L.len);
            GetMetric(t.style, ch, dx, x, y, asc, dsc, pat);
            INC(L.wid, dx);
            IF asc > L.asc THEN L.asc := asc END;
            IF dsc > L.dsc THEN L.dsc := dsc END
        END;
        L.eol := ch = EOL
    END MeasureLine;
```

2 ☞

```
    PROCEDURE DrawLine (t: Texts0.Text; len, left, right, base: INTEGER);
        VAR ch: CHAR; dx, x, y, w, h: INTEGER; pat: OS.Pattern;
    BEGIN
        WHILE len > 0 DO t.Read(ch); DEC(len);
            IF t.style.elem = NIL THEN
                OS.GetCharMetric(t.style.fnt, ch, dx, x, y, w, h, pat);
                IF left + dx < right THEN OS.DrawPattern(pat, left+x, base+y) END
            ELSE dx := t.style.elem.w;
                IF left + dx < right THEN t.style.elem.Draw(left, base) END
            END;
            INC(left, dx)
        END
    END DrawLine;
```

Frame-Methoden

```
    PROCEDURE (f: Frame) FlipCaret;
    BEGIN
        OS.DrawPattern(OS.Caret, f.caret.x, f.caret.y - 10)
    END FlipCaret;

    PROCEDURE (f: Frame) FlipSelection (a, b: Position);
        VAR x, y: INTEGER; L: Line;
    BEGIN
        L := a.L; x := a.x; y := a.y - L.dsc;
        WHILE L # b.L DO
            OS.InvertBlock(x, y, f.x + f.w - x, L.asc + L.dsc);
            L := L.next; x := f.x + scrollW + f.margin; y := y - f.lsp - L.asc - L.dsc
        END;
        OS.InvertBlock(x, y, b.x - x, L.asc + L.dsc)
    END FlipSelection;
```

3 ☞

```
    PROCEDURE (f: Frame) RedrawFrom (top: INTEGER);
        VAR t: Texts0.Text; L, L0: Line; y: INTEGER; org: LONGINT;
    BEGIN
        (*find first line to be redrawn*)
        y := f.y + f.h - f.margin; org := f.org; L0 := f.lines; L := L0.next;
        WHILE (L # f.lines) & (y - L.asc - L.dsc >= top) DO
            DEC(y, L.asc + L.dsc + f.lsp); org := org + L.len; L0 := L; L := L.next
        END;
        IF y > top THEN top := y END;
        OS.FadeCursor; OS.EraseBlock(f.x, f.y, f.w, top - f.y);
        IF f.margin > 0 THEN (*draw scroll bar*)
            OS.InvertBlock(f.x + scrollW, f.y, 1, top - f.y)
        END;
        (*redraw lines and rebuild line descriptors; L0 is last valid line descriptor*)
```

```
    t := f.text;
    LOOP NEW(L);
        t.SetPos(org); MeasureLine(t, L);
        IF (L.len = 0) OR (y - L.asc - L.dsc < f.y + f.margin) THEN EXIT END;
        t.SetPos(org);
        DrawLine(t, L.len, f.x + scrollW + f.margin, f.x + f.w - f.margin, y - L.asc);
        org := org + L.len;
        DEC(y, L.asc + L.dsc + f.lsp); L0.next := L; L0 := L;
        IF t.pos >= t.len THEN EXIT END
    END;
    L0.next := f.lines
END RedrawFrom;

PROCEDURE (f: Frame) GetPointPos (x0, y0: INTEGER; VAR p: Position);          4 ☞
    VAR t: Texts0.Text; ch: CHAR; L: Line; dx, x, y, asc, dsc: INTEGER;
        pat: OS.Pattern;
BEGIN
    (*find line containing y0*)
    L := f.lines.next; p.y := f.y + f.h - f.margin; p.org := f.org;
    WHILE (L # f.lines) & (y0 < p.y - L.asc - L.dsc - f.lsp) & L.eol DO
        DEC(p.y, L.asc + L.dsc + f.lsp); p.org := p.org + L.len; L := L.next
    END;
    DEC(p.y, L.asc);
    (*find character containig x0*)
    p.x := f.x + scrollW + f.margin; p.L := L; p.pos := p.org;
    t := f.text; t.SetPos(p.pos);
    LOOP
        IF p.pos >= t.len THEN p.dx := 0; EXIT END;
        t.Read(ch); GetMetric(t.style, ch, dx, x, y, asc, dsc, pat);
        IF (ch = EOL) OR (p.x + dx > x0) THEN p.dx := dx; EXIT
        ELSE INC(p.pos); INC(p.x, dx)
        END;
    END
END GetPointPos;

PROCEDURE (f: Frame) GetCharPos (pos: LONGINT; VAR p: Position);              5 ☞
    VAR t: Texts0.Text; ch: CHAR; L: Line; dx, x, y, asc, dsc: INTEGER;
        pat: OS.Pattern; i: LONGINT;
BEGIN
    (*find line containing pos*)
    L := f.lines.next; p.y := f.y + f.h - f.margin; p.org := f.org; p.pos := pos;
    WHILE (L # f.lines) & (pos >= p.org + L.len) & L.eol DO
        p.org := p.org + L.len; DEC(p.y, L.asc + L.dsc + f.lsp); L := L.next
    END;
    DEC(p.y, L.asc); p.L := L;
    (*find character at pos*)
    p.x := f.x + scrollW + f.margin; t := f.text; t.SetPos(p.org);
    FOR i := 1 TO p.pos - p.org DO
        t.Read(ch); GetMetric(t.style, ch, dx, x, y, asc, dsc, pat);
        INC(p.x, dx)
    END;
    IF t.pos >= t.len THEN p.dx := 0
    ELSE t.Read(ch); GetMetric(t.style, ch, p.dx, x, y, asc, dsc, pat)
```

```
    END
END GetCharPos;

PROCEDURE (f: Frame) CallCommand;
    VAR x, y, i: INTEGER; buttons: SET; p: Position; t: Texts0.Text;
        ch: CHAR; cmd: ARRAY 64 OF CHAR;
BEGIN
    REPEAT OS.GetMouse(buttons, x, y) UNTIL buttons = {};
    f.GetPointPos(x, y, p); t := f.text; t.SetPos(p.org); t.Read(ch);
    REPEAT
        WHILE (t.pos < t.len) & (ch # EOL)
        & ((CAP(ch) < "A") OR (CAP(ch) > "Z")) DO
            t.Read(ch)
        END;
        i := 0;
        WHILE (CAP(ch) >= "A") & (CAP(ch) <= "Z")
        OR (ch >= "0") & (ch <= "9") OR (ch = ".") DO
            cmd[i] := ch; INC(i); t.Read(ch)
        END;
        cmd[i] := 0X;
    UNTIL (t.pos >= t.len) OR (ch = EOL) OR (t.pos > p.pos);
    cmdFrame := f; cmdPos := t.pos; OS.Call(cmd)
END CallCommand;

PROCEDURE (f: Frame) RemoveCaret*;
BEGIN
    IF f.caret.pos >= 0 THEN f.FlipCaret; f.caret.pos := -1 END
END RemoveCaret;

PROCEDURE (f: Frame) SetCaret* (pos: LONGINT);
    VAR p: Position;
BEGIN
    IF pos < 0 THEN pos := 0 ELSIF pos > f.text.len THEN pos := f.text.len END;
    f.SetFocus; f.GetCharPos(pos, p);
    IF p.x < f.x + f.w - f.margin THEN f.caret := p; f.FlipCaret END
END SetCaret;

PROCEDURE (f: Frame) RemoveSelection*;
BEGIN
    IF f.selBeg.pos >= 0 THEN
        f.FlipSelection(f.selBeg, f.selEnd); f.selBeg.pos := -1
    END
END RemoveSelection;

PROCEDURE (f: Frame) SetSelection* (from, to: LONGINT);
BEGIN
    f.RemoveSelection;
    f.GetCharPos(from, f.selBeg); f.GetCharPos(to, f.selEnd);
    f.FlipSelection(f.selBeg, f.selEnd); f.selTime := OS.Time()
END SetSelection;

PROCEDURE (f: Frame) Defocus*;
BEGIN
```

```
    f.RemoveCaret; f.Defocus^
END Defocus;

PROCEDURE (f: Frame) Neutralize*;
BEGIN
    f.RemoveCaret; f.RemoveSelection
END Neutralize;

PROCEDURE (f: Frame) Draw*;
BEGIN
    f.RedrawFrom(f.y + f.h)
END Draw;

PROCEDURE (f: Frame) Modify* (dy: INTEGER);
    VAR y: INTEGER;
BEGIN
    y := f.y; f.Modify^ (dy);
    IF y > f.y THEN f.RedrawFrom(y) ELSE f.RedrawFrom(f.y) END
END Modify;

PROCEDURE (f: Frame) HandleMouse* (x, y: INTEGER; buttons: SET);          7 ☞
    VAR p: Position; b: SET; t: Texts0.Text; ch: CHAR; f1: Frame;
BEGIN
    t := f.text;
    IF (x < f.x + scrollW) & (buttons # {}) THEN (*handle click in scroll bar*)
        REPEAT OS.GetMouse(b, x, y); buttons := buttons + b UNTIL b = {};
        f.Neutralize;
        IF OS.left IN buttons THEN f.GetPointPos(x, y, p); f.org := p.org
        ELSIF OS.right IN buttons THEN f.org := 0
        ELSIF OS.middle IN buttons THEN
            t.SetPos((f.y + f.h - y) * f.text.len DIV f.h);
            REPEAT t.Read(ch) UNTIL (ch = EOL) OR (t.pos >= t.len);
            f.org := t.pos
        END;
        f.RedrawFrom(f.y + f.h)
    ELSE (*handle click in text area*)
        f.GetPointPos(x, y, p);
        IF OS.left IN buttons THEN
            IF p.pos # f.caret.pos THEN f.SetCaret(p.pos) END
        ELSIF OS.middle IN buttons THEN t.SetPos(p.pos); t.Read(ch);
            IF t.style.elem = NIL THEN f.CallCommand
            ELSE t.style.elem.HandleMouse(f, x, y)
            END
        ELSIF OS.right IN buttons THEN f.RemoveSelection;
            f.selBeg := p; f.selEnd := p; f.selTime := OS.Time();
            LOOP
                OS.GetMouse(b, x, y); buttons := buttons + b;
                IF b = {} THEN EXIT END;
                OS.DrawCursor(x, y); f.GetPointPos(x, y, p);
                IF p.pos < f.selBeg.pos THEN p := f.selBeg END;
                IF p.pos < t.len THEN INC(p.pos); INC(p.x, p.dx) END;
                IF p.pos # f.selEnd.pos THEN
                    IF p.pos > f.selEnd.pos THEN f.FlipSelection(f.selEnd, p)
```

```
                    ELSE f.FlipSelection(p, f.selEnd)
                    END;
                    f.selEnd := p
                END
            END;
            (*check for right-left or right-middle click*)
            IF OS.left IN buttons THEN
                t.Delete(f.selBeg.pos, f.selEnd.pos)
            ELSIF (OS.middle IN buttons)
            & (Viewers0.focus # NIL) & (Viewers0.focus IS Frame) THEN
                f1 := Viewers0.focus(Frame);
                IF f1.caret.pos >= 0 THEN
                    f1.text.Insert(f1.caret.pos, t, f.selBeg.pos, f.selEnd.pos)
                END
            END
        END
    END
END HandleMouse;

PROCEDURE (f: Frame) HandleKey* (ch: CHAR);
    VAR pos: LONGINT;
BEGIN
    pos := f.caret.pos;
    IF pos >= 0 THEN
        IF ch = DEL THEN
            IF pos > 0 THEN f.text.Delete(pos - 1, pos); f.SetCaret(pos - 1) END
        ELSE f.text.SetPos(pos); f.text.Write(ch); f.SetCaret(pos + 1)
        END
    END
END HandleKey;

8 ☞ PROCEDURE (f: Frame) Handle* (VAR m: OS.Message);
    VAR t: Texts0.Text; ch: CHAR; VAR dx, x, y, asc, dsc: INTEGER;
        pat: OS.Pattern; p: Position;
BEGIN
    t := f.text;
    WITH
        m: Texts0.NotifyInsMsg DO
            IF m.t = t THEN
                IF m.beg < f.org THEN f.org := f.org + (m.end - m.beg)
                ELSE
                    f.Neutralize; OS.FadeCursor;
                    f.GetCharPos(m.beg, p);
                    t.SetPos(m.beg); t.Read(ch);
                    GetMetric(t.style, ch, dx, x, y, asc, dsc, pat);
                    IF (m.end = m.beg+1) & (ch # EOL) & (p.L # f.lines)
                    & (asc+dsc <= p.L.asc+p.L.dsc) THEN
                        IF p.x + dx <= f.x + f.w - f.margin THEN
                            OS.CopyBlock(p.x, p.y-p.L.dsc, f.x+f.w-f.margin-dx-p.x,
                                p.L.asc+p.L.dsc, p.x+dx, p.y-p.L.dsc);
                            OS.EraseBlock(p.x, p.y-p.L.dsc, dx, p.L.asc + p.L.dsc);
                            IF t.style.elem = NIL THEN
                                OS.DrawPattern(pat, p.x + x, p.y + y)
```

```
                    ELSE t.style.elem.Draw(p.x, p.y)
                    END
                ELSE
                    OS.EraseBlock(p.x, p.y-p.L.dsc, f.x+f.w-p.x,
                        p.L.asc+p.L.dsc)
                END;
                INC(p.L.len); INC(p.L.wid, dx)
            ELSE f.RedrawFrom(p.y + p.L.asc)
            END
        END
    END
    | m: Texts0.NotifyDelMsg DO
        IF m.t = t THEN
            IF m.end <= f.org THEN f.org := f.org - (m.end - m.beg)
            ELSE
                f.Neutralize;
                IF m.beg < f.org THEN f.org := m.beg; f.RedrawFrom(f.y + f.h)
                ELSE f.GetCharPos(m.beg, p); f.RedrawFrom(p.y + p.L.asc)
                END
            END
        END
    | m: Texts0.NotifyReplMsg DO
        IF (m.t = t) & (m.end > f.org) THEN
            f.Neutralize;
            IF m.beg < f.org THEN m.beg := f.org END;
            f.GetCharPos(m.beg, p); f.RedrawFrom(p.y + p.L.asc)
        END
    | m: SelectionMsg DO
        IF (f.selBeg.pos >= 0) & ((m.f = NIL)
        OR (m.f.selTime < f.selTime)) THEN
            m.f := f
        END
    ELSE
    END
END Handle;

PROCEDURE New* (t: Texts0.Text): Frame;
    VAR f: Frame; fnt: OS.Font;
BEGIN
    NEW(f); f.text := t;
    f.org := 0; f.caret.pos := -1; f.selBeg.pos := -1; f.lsp := 2; f.margin := 5;
    NEW(f.lines); f.lines.next := f.lines; fnt := OS.DefaultFont();
    f.lines.asc := fnt.maxY; f.lines.dsc := - fnt.minY; f.lines.len := 0;
    RETURN f
END New;

PROCEDURE NewMenu* (name, menu: ARRAY OF CHAR): Frame;
    VAR t: Texts0.Text; f: Frame; i: INTEGER;
BEGIN
    NEW(t); t.Clear;
    i := 0; WHILE name[i] # 0X DO t.Write(name[i]); INC(i) END;
    t.Write(" "); t.Write("|"); t.Write(" ");
    i := 0; WHILE menu[i] # 0X DO t.Write(menu[i]); INC(i) END;
```

```
    f := New(t); f.margin := 0; RETURN f
END NewMenu;

PROCEDURE (f: Frame) Copy* (): Viewers0.Frame;
    VAR f1: Frame;
BEGIN
    f1 := New(f.text); f1.margin := f.margin; RETURN f1
END Copy;

PROCEDURE GetSelection* (VAR f: Frame);
    VAR m: SelectionMsg;
BEGIN
    m.f := NIL; Viewers0.Broadcast(m); f := m.f
END GetSelection;

END TextFrames0.
```

9 ☞

1 ☞ *MeasureLine* liest die Zeile von der momentanen Textposition bis zum nächsten Zeilenende und liefert einen Zeilendeskriptor mit den Maßen gemäß Abb. 12.23. Die Metrik jedes einzelnen Zeichens wird mittels *GetMetric* angefordert.

2 ☞ *DrawLine* liest *len* Zeichen ab der momentanen Textposition und stellt sie auf dem Bildschirm dar. *left* ist der linke, *right* der rechte Zeilenrand, *base* die Höhe der Basislinie in Pixel relativ zur linken unteren Ecke des Rahmens. Zeichen, die über den rechten Rand hinausgehen, werden nicht dargestellt (*clipping*). Elemente werden aufgefordert, sich selbst darzustellen, da der Rahmen nicht wissen kann, wie sie zu zeichnen sind.

3 ☞ *RedrawFrom* zeichnet alle Zeilen ab der vertikalen Pixelposition *top* und erstellt neue Zeilendeskriptoren für sie. *y* zeigt jeweils auf den oberen Rand der zu zeichnenden Zeile und *org* auf den Index des ersten Zeichens darin. Bevor eine Zeile gezeichnet wird, wird sie mit *MeasureLine* vermessen. Der Abstand zwischen zwei Zeilen ist in unserer Implementierung immer *f.lsp*. Falls *f* kein Menürahmen ist (*f.margin* > 0), wird auch ein Rollbalken gezeichnet.

4 ☞ *GetPointPos* berechnet den Positionsdeskriptor *p* des Zeichens auf dem Bildschirm, das den Punkt (*x0*, *y0*) enthält oder ihm am nächsten kommt (siehe Abb. 12.20).

5 ☞ *GetCharPos* berechnet den Positionsdeskriptor *p* des Zeichens an der Textposition *pos* (siehe Abb. 12.20).

6 ☞ Wenn der Benutzer mit der mittleren Maustaste in den Text klickt, wird das Wort an dieser Stelle als Oberon-Kommando interpretiert und mittels *OS.Call* aufgerufen.

7 ☞ Klickt man mit der Maus in einen Textrahmen, führt das zum Aufruf der Methode *HandleMouse*. (*x*, *y*) ist die Mausposition und *buttons* die Menge der gedrückten Mausknöpfe. Wenn (*x*, *y*) im Rollbalken liegt, wird geblättert; wenn der Punkt im Textbereich liegt, wird in

Abhängigkeit vom gedrückten Mausknopf das Caret gesetzt, ein Textstück selektiert oder ein Kommando ausgeführt. Wird mit dem mittleren Mausknopf auf ein Element geklickt, reagiert nicht der Rahmen, sondern der Klick wird dem Element zur Behandlung übergeben. So können Elemente, die dem Rahmen unbekannt sind, nach ihrer Art auf den Klick reagieren. Beim Setzen einer Selektion wird eine Zeitmarke gespeichert.

Die meisten Meldungen an Textrahmen sind als Methoden implementiert: ihr Adressat ist bekannt. Bei einigen jedoch (z.B. bei *Notify*-Meldungen) ist der Adressat unbekannt. Darum müssen sie an *alle* Textrahmen geschickt werden, wobei es jedem Textrahmen überlassen bleibt, herauszufinden, ob die Meldung für ihn bestimmt ist oder nicht. Solche Meldungen werden nicht als Methoden, sondern als Meldungsobjekte implementiert. *Handle* ist der entsprechende Meldungsinterpreter. *8* ☞

Jede Änderung in einem Text führt zum Versenden einer *Notify*-Meldung an alle Rahmen am Bildschirm. Durch die *Notify*-Meldung werden diejenigen Rahmen, die den geänderten Text enthalten, aufgefordert, die Änderung am Bildschirm nachzuführen (Abb. 11.10).

NotifyInsMsg bedeutet, daß etwas in den Text eingefügt wurde. Die Meldung wird vom Rahmen *f* behandelt, wenn der eigene Text *f.text* dem geänderten Text *m.t* entspricht. In der vorliegenden Implementierung wurde nur der Fall optimiert, in dem ein einzelnes Zeichen eingefügt (eingetippt) wurde. In allen anderen Fällen wird der Rahmen-Inhalt ab der Einfügeposition neu aufgebaut.

NotifyDelMsg bedeutet, daß im Text etwas gelöscht wurde. *NotifyReplMsg* bedeutet, daß sich im Text etwas geändert hat, ohne daß die Länge des Texts dabei verändert wurde. Aus Einfachheitsgründen wird bei *NotifyDelMsg* und *NotifyReplMsg* immer der ganze Rahmen-Inhalt ab der geänderten Position neu aufgebaut. Im Oberon-System sind diese Operationen optimiert, so daß möglichst wenig neu gezeichnet werden muß. Das ist aber kompliziert und wurde hier weggelassen.

Handle interpretiert schließlich noch *SelectionMsg* (siehe unten): Wenn die Selektion von *f* jünger ist als die von *m.f*, wird *m.f* durch *f* ersetzt.

GetSelection bestimmt die jüngste Selektion in allen sichtbaren Textrahmen. Zu diesem Zweck wird ein Meldungsobjekt vom Typ *SelectionMsg* an alle Rahmen geschickt. Textrahmen reagieren darauf, indem sie sich in dieses Record eintragen, falls in ihnen eine Selektion gesetzt ist, die jünger ist, als die bisher jüngste Selektion. *9* ☞

Was kann man aus dieser Implementierung lernen?

Aus objektorientierter Sicht sind an *TextFrames0* vor allem drei Dinge interessant:

1. *Rahmenliste als heterogene Liste in Fenstern.* Ein Textrahmen kann in ein Fenster eingehängt werden und wird von diesem richtig behandelt, obwohl Fenster Textrahmen nicht kennen. Fenster arbeiten mit abstrakten Rahmen, hinter denen unter anderem ein Textrahmen stehen kann.

2. *Unterstützung des MVC-Schemas.* Ein Textrahmen übernimmt die Rolle des Eingabeteils und der Sicht eines Texteditors. Bei Änderung des Textes wird ein Meldungsobjekt an alle Rahmen geschickt. Der Meldungsinterpreter *Handle* zeigt, wie Textrahmen darauf reagieren. Die Verteilung einer Meldung an mehrere Empfänger ist die Hauptanwendung von Meldungsobjekten. Sie kann an *TextFrames0* studiert werden.

3. *Behandlung von Elementen in Texten.* Textrahmen müssen Elemente am Bildschirm darstellen und manipulieren. Da sie nicht wissen, welche Arten von Elementen es gibt, arbeiten sie mit einer abstrakten Klasse *Element*, hinter der jede beliebige Elementart stecken kann.

12.3.4 Hauptmodul des Texteditors (Edit0)

Nun fehlt nur noch ein Hauptmodul, das einen Textrahmen öffnet und in ein Fenster einhängt sowie verschiedene andere Benutzerkommandos ausführt. Diese Aufgabe wird vom Modul *Edit0* übernommen, das drei Kommandos anbietet:

- *Edit0.Open name* öffnet ein Fenster mit einem Textrahmen und zeigt darin die Textdatei namens *name* an.
- *Edit0.Store* speichert den Inhalt des Textrahmens, zu dessen Menü dieses Kommando gehört, auf eine Datei ab. Der Name der Datei entspricht dem Namen des Fensters, das den Textrahmen enthält (Der Name des Fensters steht am Anfang des dazugehörigen Menürahmens).
- *Edit0.ChangeFont fontname* ändert die Schriftart des gerade selektierten Textstücks auf die Schriftart namens *fontname*.

Zum Lesen der Kommandoparameter benutzt *Edit0* das Modul *In*, das in Anhang B beschrieben wird.

```
MODULE Edit0;
IMPORT OS, In, TextFrames0, Texts0, Viewers0;

PROCEDURE Open*;
    VAR t: Texts0.Text; menu, cont: TextFrames0.Frame;
        v: Viewers0.Viewer; f: OS.File; r: OS.Rider;
        fileName: ARRAY 32 OF CHAR;
BEGIN
    In.Open; In.Name(fileName);
    IF In.Done THEN
        menu := TextFrames0.NewMenu(fileName,
            "Viewers0.Close  Viewers0.Copy  Edit0.Store");
        NEW(t); f := OS.OldFile(fileName);
        IF f = NIL THEN t.Clear
        ELSE OS.InitRider(r); r.Set(f, 0); t.Load(r)
        END;
        cont := TextFrames0.New(t);
        v := Viewers0.New(menu, cont)
    END
END Open;

PROCEDURE Store*;
    VAR v: Viewers0.Viewer; f: OS.File; r: OS.Rider;
    name: ARRAY 32 OF CHAR;
BEGIN
    v := Viewers0.ViewerAt(TextFrames0.cmdFrame.y);
    In.OpenText(v.menu(TextFrames0.Frame).text, 0);
    In.Name(name); (*read viewer name*)
    f := OS.NewFile(name); OS.InitRider(r); r.Set(f, 0);
    v.Neutralize; v.cont(TextFrames0.Frame).text.Store(r);
    OS.Register(f)
END Store;

PROCEDURE ChangeFont*;
    VAR f: TextFrames0.Frame; fontName: ARRAY 32 OF CHAR;
BEGIN
    In.Open; In.Name(fontName);
    TextFrames0.GetSelection(f);
    IF (f # NIL) & In.Done THEN
        f.text.ChangeFont(f.selBeg.pos, f.selEnd.pos,
            OS.FontWithName(fontName))
    END
END ChangeFont;

END Edit0.
```

12.4 Ein Grafikeditor

Neben Textfenstern wollen wir auch Fenster haben, in denen man Grafik editieren kann. Es soll möglich sein, verschiedene Figuren, wie Rechtecke, Linien oder Kreise zu zeichnen, zu verschieben, zu selektieren und zu löschen.

Auch ein Grafikeditor ist ein interaktives Programm, das man nach dem MVC-Schema aufbaut. Die Modell-Komponente ist hier eine Grafik vom Typ *Shapes0.Graphic*, die eine Liste von Figuren vom Typ *Shapes0.Shape* verwaltet. Die Sicht und der Eingabeteil werden von der Klasse *GraphicFrames0.Frame* wahrgenommen, die in einem Fenster vom Typ *Viewers0.Viewer* installiert werden kann. Das Hauptmodul heißt *Draw0* (Abb. 12.25).

Auch hier gilt: wann immer das Modell geändert wird, indem man eine Figur erzeugt, löscht oder selektiert, müssen alle Sichten davon informiert werden. Das geschieht wie beim Texteditor über *Notify*-Meldungen, die das Modell an alle Sichten schickt.

12.4.1 Figuren (Shapes0)

Das Modul *Shapes0* verwaltet eine Grafik (Klasse *Graphic*) als das Datenmodell des Grafikeditors. So wie ein Text aus Zeichen und

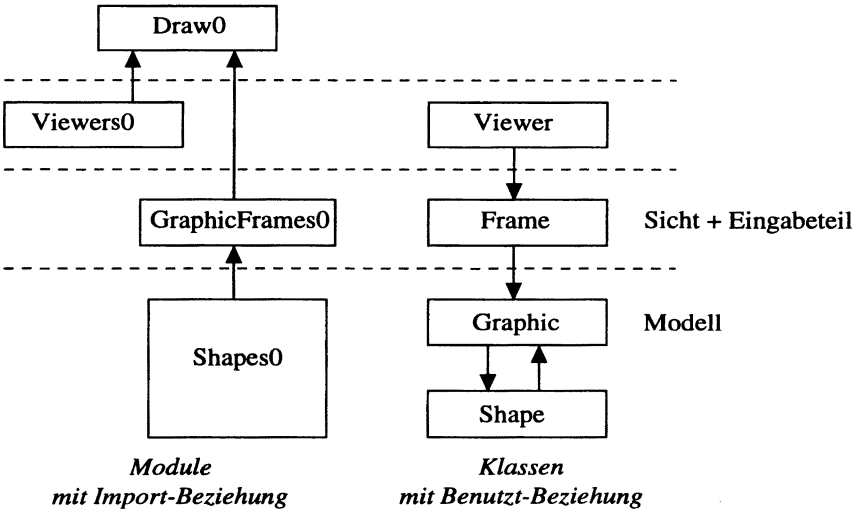


Abb. 12.25 Module und Klassen des Grafikeditors

Elementen besteht, besteht eine Grafik aus Figuren. Eine Grafik soll *beliebige* Figuren enthalten können, auch solche, die erst zu einem späteren Zeitpunkt definiert werden. Daher darf *Graphic* die Figurenarten nicht kennen, sondern muß mit einer abstrakten Klasse *Shape* arbeiten, von der spätere Figurenarten abgeleitet sind (Abb. 12.26).

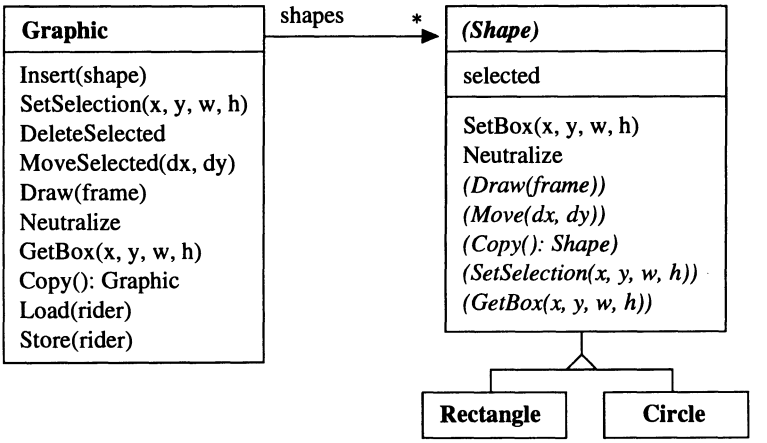


Abb. 12.26 Grafiken und ihre Figuren

Die Schnittstelle von *Shapes0* sieht folgendermaßen aus:

Schnittstelle von Shapes0

```
DEFINITION Shapes0;
IMPORT OS, Viewers0;

TYPE
   Shape = POINTER TO ShapeDesc;
   ShapeDesc = RECORD (OS.ObjectDesc)
      selected: BOOLEAN;          (*TRUE: shape is selected*)
      PROCEDURE (s: Shape) SetBox (x, y, w, h: INTEGER);
      PROCEDURE (s: Shape) GetBox (VAR x, y, w, h: INTEGER);
      PROCEDURE (s: Shape) Draw (f: Viewers0.Frame);
      PROCEDURE (s: Shape) Move (dx, dy: INTEGER);
      PROCEDURE (s: Shape) Neutralize;
      PROCEDURE (s: Shape) SetSelection (x, y, w, h: INTEGER);
      PROCEDURE (s: Shape) Copy (): Shape;
   END;

   Graphic = POINTER TO GraphicDesc;
   GraphicDesc = RECORD
      shapes: Shape;
      PROCEDURE (g: Graphic) Insert (s: Shape);
      PROCEDURE (g: Graphic) SetSelection (x, y, w, h: INTEGER);
      PROCEDURE (g: Graphic) DeleteSelected;
      PROCEDURE (g: Graphic) MoveSelected (dx, dy: INTEGER);
      PROCEDURE (g: Graphic) Draw (f: Viewers0.Frame);
```

```
        PROCEDURE (g: Graphic) Neutralize;
        PROCEDURE (g: Graphic) GetBox (VAR x, y, w, h: INTEGER);
        PROCEDURE (g: Graphic) Copy (): Graphic;
        PROCEDURE (g: Graphic) Load (VAR r: OS.Rider);
        PROCEDURE (g: Graphic) Store (VAR r: OS.Rider);
    END ;

    NotifyChangeMsg = RECORD (OS.Message)
        g: Graphic
    END ;

VAR
    curShape: Shape; (*prototype object for creating new shapes*)

PROCEDURE InitGraphic (VAR g: Graphic);

END Shapes0.
```

Shape-Meldungen

- *s.SetBox(x, y, w, h)* berechnet Position, Form und Größe der Figur *s* auf Grund des umschreibenden Rechtecks *(x, y, w, h)*.
- *s.GetBox(x, y, w, h)* liefert das kleinste Rechteck *(x, y, w, h)*, das *s* einschließt.
- *s.Draw(f)* zeichnet die Figur *s* an ihrer momentanen Position im Rahmen *f*.
- *s.Move(dx, dy)* verschiebt s um den Vektor *(dx, dy)*.
- *s.Neutralize* löscht eine eventuelle Selektion von *s*.
- *s.SetSelection(x, y, w, h)* selektiert die Figur *s*, falls sie im Rechteck *(x, y, w, h)* liegt.
- *s1 := s.Copy()* liefert eine Kopie von *s*.

Graphic-Meldungen

- *g.Insert (s)* fügt die Figur *s* in die Grafik *g* ein.
- *g.SetSelection (x, y, w, h)* selektiert alle Figuren der Grafik *g*, die im Rechteck *(x, y, w, h)* liegen.
- *g.DeleteSelected* löscht alle selektierten Figuren in *g*.
- *g.MoveSelected (dx, dy)* verschiebt alle selektierten Figuren in *g* um den Vektor *(dx, dy)*.
- *g.Draw (f)* fordert alle Figuren in *g* auf, sich an ihrer Position im Rahmen *f* zu zeichnen.
- *g.Neutralize* entfernt eine eventuelle Selektion von Figuren in *g*.
- *g.GetBox (x, y, w, h)* liefert das kleinste Rechteck, das alle Figuren in der Grafik *g* umschließt.
- *g1 := g.Copy()* liefert eine Kopie von *g*.
- *g.Load (r)* lädt die Grafik *g* vom Rider *r*.
- *g.Store (r)* speichert die Grafik *g* auf den Rider *r*.

Implementierung von Shapes0

```
MODULE Shapes0;
IMPORT OS, Viewers0;

TYPE
   Shape* = POINTER TO ShapeDesc;
   ShapeDesc* = RECORD (OS.ObjectDesc)
      selected*: BOOLEAN;  (*TRUE: shape is selected*)
      next: Shape
   END;

   Graphic* = POINTER TO GraphicDesc;
   GraphicDesc* = RECORD
      shapes*: Shape
   END;

   NotifyChangeMsg* = RECORD (OS.Message) g*: Graphic END;

VAR
   curShape*: Shape;  (*prototype object for creating new shapes*)
```

Shape-Methoden

```
PROCEDURE (s: Shape) SetBox* (x, y, w, h: INTEGER);
BEGIN
   s.selected := FALSE;
END SetBox;

PROCEDURE (s: Shape) Draw* (f: Viewers0.Frame);
BEGIN HALT(99)  (*abstract*)
END Draw;

PROCEDURE (s: Shape) Move* (dx, dy: INTEGER);
BEGIN HALT(99)  (*abstract*)
END Move;

PROCEDURE (s: Shape) SetSelection* (x, y, w, h: INTEGER);
BEGIN HALT(99)  (*abstract*)
END SetSelection;

PROCEDURE (s: Shape) Neutralize*;
BEGIN
   s.selected := FALSE
END Neutralize;

PROCEDURE (s: Shape) GetBox* (VAR x, y, w, h: INTEGER);
BEGIN HALT(99)  (*abstract*)
END GetBox;

PROCEDURE (s: Shape) Copy* (): Shape;
BEGIN HALT(99)  (*abstract*)
END Copy;
```

Graphic-Methoden

```
PROCEDURE InitGraphic* (VAR g: Graphic);
BEGIN g.shapes := NIL
END InitGraphic;
```

```
PROCEDURE (g: Graphic) Insert* (s: Shape);
    VAR msg: NotifyChangeMsg;
BEGIN
    s.next := g.shapes; g.shapes := s; msg.g := g; Viewers0.Broadcast(msg)
END Insert;

PROCEDURE (g: Graphic) DeleteSelected*;
    VAR s, s0: Shape; msg: NotifyChangeMsg;
BEGIN
    s := g.shapes; s0 := NIL;
    WHILE s # NIL DO
        IF s.selected THEN
            IF s0 = NIL THEN g.shapes := s.next ELSE s0.next := s.next END
        ELSE s0 := s
        END;
        s := s.next
    END;
    msg.g := g; Viewers0.Broadcast(msg)
END DeleteSelected;

PROCEDURE (g: Graphic) MoveSelected* (dx, dy: INTEGER);
    VAR s: Shape; msg: NotifyChangeMsg;
BEGIN
    s := g.shapes;
    WHILE s # NIL DO
        IF s.selected THEN s.Move(dx, dy) END;
        s := s.next
    END;
    msg.g := g; Viewers0.Broadcast(msg)
END MoveSelected;

PROCEDURE (g: Graphic) Draw* (f: Viewers0.Frame);
    VAR s: Shape;
BEGIN
    s := g.shapes; WHILE s # NIL DO s.Draw(f); s := s.next END
END Draw;

PROCEDURE (g: Graphic) Neutralize*;
    VAR s: Shape; msg: NotifyChangeMsg; changed: BOOLEAN;
BEGIN
    s := g.shapes; changed := FALSE;
    WHILE s # NIL DO
        changed := changed OR s.selected; s.Neutralize; s := s.next
    END;
    IF changed THEN msg.g := g; Viewers0.Broadcast(msg) END
END Neutralize;

PROCEDURE (g: Graphic) SetSelection* (x, y, w, h: INTEGER);
    VAR s: Shape; msg: NotifyChangeMsg;
BEGIN
    s := g.shapes;
    WHILE s # NIL DO s.SetSelection(x, y, w, h); s := s.next END;
    msg.g := g; Viewers0.Broadcast(msg)
```

```
END SetSelection;

PROCEDURE (g: Graphic) GetBox* (VAR x, y, w, h: INTEGER);
    VAR x0, y0, w0, h0: INTEGER; s: Shape;
BEGIN
    x := 0; y := 0; w := 12; h := 12;
    s := g.shapes;
    IF s # NIL THEN s.GetBox(x, y, w, h); s := s.next END;
    WHILE s # NIL DO
        s.GetBox(x0, y0, w0, h0);
        IF x0 < x THEN INC(w, x - x0); x := x0 END;
        IF y0 < y THEN INC(h, y - y0); y := y0 END;
        IF x0 + w0 > x + w THEN w := x0 + w0 - x END;
        IF y0 + h0 > y + h THEN h := y0 + h0 - y END;
        s := s.next
    END;
END GetBox;

PROCEDURE (g: Graphic) Copy* (): Graphic;
    VAR s, a, b: Shape; g1: Graphic;
BEGIN
    NEW(g1); g1.shapes := NIL;
    s := g.shapes;
    WHILE s # NIL DO
        a := s.Copy(); a.next := NIL;
        IF g1.shapes = NIL THEN g1.shapes := a ELSE b.next := a END;
        b := a; s := s.next
    END;
    RETURN g1
END Copy;

PROCEDURE (g: Graphic) Load* (VAR r: OS.Rider);
    VAR s, last: Shape; x: OS.Object;
BEGIN
    last := NIL;
    REPEAT
        r.ReadObj(x);
        IF x = NIL THEN s := NIL ELSE s := x(Shape) END;
        IF last = NIL THEN g.shapes := s ELSE last.next := s END;
        last := s
    UNTIL x = NIL  (*terminated by a NIL shape*)
END Load;

PROCEDURE (g: Graphic) Store* (VAR r: OS.Rider);
    VAR s: Shape;
BEGIN
    s := g.shapes; WHILE s # NIL DO r.WriteObj(s); s := s.next END;
    r.WriteObj(NIL)
END Store;

BEGIN
    curShape := NIL
END Shapes0.
```

12.4.2 Editieren von Figuren (GraphicFrames0)

Ein Grafikrahmen stellt Figuren am Bildschirm dar und interpretiert Mausklicks, indem er Figuren erzeugt, verschiebt, selektiert oder löscht. Um das Beispiel klein zu halten, kann man die Größe von Figuren in unseren Grafikrahmen nicht ändern.

Die Mausknöpfe haben in Grafikrahmen folgende Bedeutung: Wenn man den linken Knopf drückt und an einer anderen Stelle wieder losläßt, wird im überstrichenen Rechteck eine neue Figur aufgespannt. Bewegt man die Maus mit gedrücktem mittlerem Knopf, wird die gesamte Zeichnung im Rahmen verschoben; drückt man gleichzeitig den linken Knopf, werden nur die selektierten Figuren verschoben. Mit dem rechten Knopf kann man schließlich selektieren. Alle Figuren im Rechteck, das mit gedrücktem rechten Mausknopf überstrichen wird, werden selektiert angezeigt; drückt man gleichzeitig den linken Knopf, werden die selektierten Figuren gelöscht.

Um die gesamte Zeichnung am Bildschirm verschieben zu können, ohne die Koordinaten sämtlicher Figuren ändern zu müssen, besitzt ein Grafikrahmen ein *Koordinatensystem* mit Ursprung (*orgX*, *orgY*) relativ zur linken unteren Ecke des Rahmens (Abb. 12.27). Die Koordinaten der Figuren sind relativ zum Ursprung des Koordinatensystems, so daß man durch Verschieben des Ursprungs die gesamte Zeichnung verschiebt.

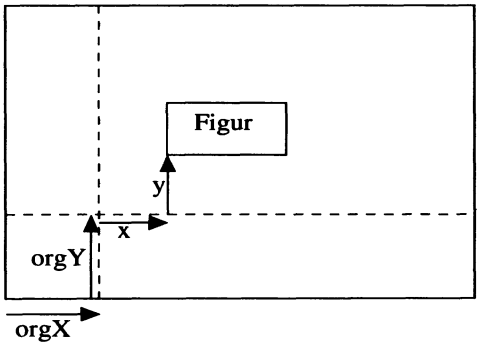


Abb. 12.27 Grafikrahmen mit Ursprung (orgX, orgY)

Grafikrahmen sind im Modul *GraphicFrames0* implementiert, das folgende Schnittstelle besitzt:

Schnittstelle von GraphicFrames0

```
DEFINITION GraphicFrames0;
IMPORT Viewers0, OS, Shapes0;

TYPE
   Frame = POINTER TO FrameDesc;
   FrameDesc = RECORD (Viewers0.FrameDesc)
      orgX, orgY: INTEGER;
      graphic: Shapes0.Graphic;
      PROCEDURE (f: Frame) Draw;
      PROCEDURE (f: Frame) Neutralize;
      PROCEDURE (f: Frame) Modify (y: INTEGER);
      PROCEDURE (f: Frame) Copy (): Viewers0.Frame;
      PROCEDURE (f: Frame) HandleMouse (x, y: INTEGER; but: SET);
      PROCEDURE (f: Frame) Handle (VAR m: OS.Message);
      PROCEDURE (f: Frame) InvertBlock (x, y, w, h: INTEGER);
   END ;

PROCEDURE New (graphic: Shapes0.Graphic): Frame;

END GraphicFrames0.
```

Der Großteil der Schnittstelle von *Frame* wird von der Basisklasse *Viewers0.Frame* geerbt. Nur eine einzige Methode ist neu:

- *f.InvertBlock(x, y, w, h)* invertiert im Rahmen *f* den Block *(x, y, w, h)*. *x* und *y* sind relativ zum Ursprung *(orgX, orgY)*. Falls der Block über den Rand des Rahmens hinausragt, wird er abgeschnitten (clipping). *InvertBlock* kann auch zum Zeichnen von horizontalen und vertikalen Linien verwendet werden (*w*=1 oder *h*=1).

Implementierung von GraphicFrames0

```
MODULE GraphicFrames0;
IMPORT OS, Viewers0, Shapes0;

TYPE
   Frame* = POINTER TO FrameDesc;
   FrameDesc* = RECORD (Viewers0.FrameDesc)
      orgX*, orgY*: INTEGER;          (*origin*)
      graphic*: Shapes0.Graphic       (*shapes in this frame*)
   END;

PROCEDURE (f: Frame) InvertBlock* (x, y, w, h: INTEGER);
BEGIN
   INC(x, f.x + f.orgX); INC(y, f.y + f.orgY);
   IF x < f.x THEN DEC(w, f.x - x); x := f.x END;
   IF x + w > f.x + f.w THEN w := f.x + f.w - x END;
   IF y < f.y THEN DEC(h, f.y - y); y := f.y END;
   IF y + h > f.y + f.h THEN h := f.y + f.h - y END;
   IF (w > 0) & (h > 0) THEN OS.InvertBlock(x, y, w, h) END
END InvertBlock;
```

1 ☞

2 ☞

```
PROCEDURE (f: Frame) Draw*;
BEGIN
   OS.FadeCursor;
   OS.EraseBlock(f.x, f.y, f.w, f.h);
   f.graphic.Draw(f)
END Draw;

PROCEDURE (f: Frame) Modify* (y: INTEGER);
BEGIN
   f.Modify^ (y); f.Draw
END Modify;
```

3 ☞

```
PROCEDURE (f: Frame) HandleMouse* (x, y: INTEGER; buttons: SET);
   VAR w, h, dx, dy: INTEGER; s: Shapes0.Shape; changed: BOOLEAN;

      PROCEDURE Track(VAR x, y, w, h, dx, dy: INTEGER;
         VAR buttons: SET);
         VAR b: SET; x1, y1: INTEGER;
      BEGIN
         REPEAT
            OS.GetMouse(b, x1, y1); buttons := buttons + b;
            OS.DrawCursor(x1, y1)
         UNTIL b = {};
         dx := x1 - x; dy := y1 - y; w := ABS(dx); h := ABS(dy);
         IF x1 < x THEN x := x1 END;
         IF y1 < y THEN y := y1 END;
         DEC(x, f.x + f.orgX); DEC(y, f.y + f.orgY)
      END Track;

BEGIN
   changed := FALSE;
   IF OS.left IN buttons THEN
      Track(x, y, w, h, dx, dy, buttons);
      (*generate new shape from prototype curShape*)
      IF Shapes0.curShape # NIL THEN
         s := Shapes0.curShape.Copy();
         s.SetBox(x, y, w, h); f.graphic.Insert(s)
      END
   ELSIF OS.middle IN buttons THEN
      Track(x, y, w, h, dx, dy, buttons);
      IF OS.left IN buttons THEN (*middle+left click: move selected shapes*)
         f.graphic.MoveSelected(dx, dy)
      ELSE (*middle click: move origin*)
         INC(f.orgX, dx); INC(f.orgY, dy); f.Draw
      END
   ELSIF OS.right IN buttons THEN
      f.Neutralize; Track(x, y, w, h, dx, dy, buttons);
      f.graphic.SetSelection(x, y, w, h);
      IF OS.left IN buttons THEN (*right+left click: delete selected shapes*)
         f.graphic.DeleteSelected
      END
   END
END HandleMouse;
```

```
PROCEDURE (f: Frame) Handle* (VAR m: OS.Message);                    4 ☞
BEGIN
   WITH m: Shapes0.NotifyChangeMsg DO
      IF f.graphic = m.g THEN f.Draw END
   ELSE
   END
END Handle;

PROCEDURE (f: Frame) Neutralize*;
BEGIN
   f.graphic.Neutralize
END Neutralize;

PROCEDURE New* (graphic: Shapes0.Graphic): Frame;
   VAR f: Frame;
BEGIN
   NEW(f); f.graphic := graphic;
   f.orgX := 0; f.orgY := 0;
   RETURN f
END New;

PROCEDURE (f: Frame) Copy* (): Viewers0.Frame;
   VAR f1: Frame;
BEGIN
   f1 := New(f.graphic); f1.orgX := f.orgX; f1.orgY := f.orgY;
   RETURN f1
END Copy;

END GraphicFrames0.
```

Ein Grafikrahmen stellt seinen Klienten üblicherweise Operationen zur Verfügung, mit denen sie im Koordinatensystem des Rahmens zeichnen können. Dabei werden die Koordinaten von Figuren (die relativ zum Ursprung *orgX*, *orgY* sind) auf Bildschirmkoordinaten umgerechnet. In diesem einfachen Beispiel gibt es nur eine einzige Zeichenoperation *InvertBlock*. Sie sorgt auch dafür, daß nicht über den Rand des Rahmens hinausgezeichnet wird (clipping). 1 ☞

Draw zeichnet den gesamten Rahmeninhalt neu. Um die Implementierung einfach zu halten, geschieht dies bei jeder Änderung im Rahmen. In der Praxis ist das natürlich nicht tragbar. Dort muß man dafür sorgen, daß bei einer Änderung nur soviel gezeichnet wird, wie unbedingt nötig ist. 2 ☞

HandleMouse interpretiert Mausklicks wie oben beschrieben. *Track* berechnet den Start- und Endpunkt einer Mausbewegung mit gedrückten Knopf. Die Koordinaten dieser Punkte werden so transformiert, daß sie relativ zum Ursprung des Rahmens sind. 3 ☞

Interessant ist die Frage, wie der Benutzer neue Figuren eingeben kann. Wenn er den linken Mausknopf drückt, muß der Rahmen als Reaktion darauf eine neue Figur erzeugen und am Bildschirm darstel-

len. Welche Figur soll das aber sein? Ein Rechteck? Ein Kreis? Der Rahmen kennt überhaupt keine Rechtecke oder Kreise, sondern nur abstrakte Figuren. Er muß sich also mit einem Trick behelfen: Er erzeugt eine Kopie des Objekts, das gerade in den globalen Variablen *Shapes0.curShape* gespeichert ist. Diese Variable enthält ein *Prototyp*-Objekt (siehe Kapitel 9.2.3), das ein Rechteck, ein Kreis oder eine andere Figur sein kann. Sie wird in Modulen gesetzt, die konkrete Figurenklassen implementieren.

4 ☞ *Handle* ist der Meldungsinterpreter von Grafikrahmen. Er interpretiert *NotifyChange*-Meldungen, die verschickt werden, wenn eine Figur verändert wird.

12.4.3 Hauptmodul des Grafikeditors (Draw0)

Draw0 stellt zwei Kommandos zur Verfügung:

- *Draw0.Open* name öffnet ein Fenster mit einem Grafikrahmen und zeigt darin den Inhalt der Grafikdatei namens *name* an.
- *Draw0.Store* speichert den Inhalt des Grafikrahmens, zu dessen Menü dieses Kommando gehört, auf eine Datei ab. Der Name der Datei entspricht dem Namen des Fensters, das den Grafikrahmen enthält.

Implementierung von Draw0

```
MODULE Draw0;
IMPORT OS, In, TextFrames0, Shapes0, GraphicFrames0, Viewers0;

PROCEDURE Open*;
   VAR v: Viewers0.Viewer; fileName: ARRAY 32 OF CHAR;
      menu: TextFrames0.Frame; cont: GraphicFrames0.Frame;
      file: OS.File; r: OS.Rider; g: Shapes0.Graphic;
BEGIN
   In.Open; In.Name(fileName);
   IF In.Done THEN
      menu := TextFrames0.NewMenu(fileName,
         "Viewers0.Close  Viewers0.Copy  Draw0.Store");
      NEW(g); Shapes0.InitGraphic(g);  file := OS.OldFile(fileName);
      IF file # NIL THEN OS.InitRider(r); r.Set(file, 0); g.Load(r) END;
      cont := GraphicFrames0.New(g);
      v := Viewers0.New(menu, cont)
   END
END Open;

PROCEDURE Store*;
   VAR v: Viewers0.Viewer; file: OS.File; r: OS.Rider;
      name: ARRAY 32 OF CHAR;
BEGIN
   v := Viewers0.ViewerAt(TextFrames0.cmdFrame.y);
```

```
    In.OpenText(v.menu(TextFrames0.Frame).text, 0);
    In.Name(name); (*read viewer name*)
    file := OS.NewFile(name); OS.InitRider(r); r.Set(file, 0);
    v.cont(GraphicFrames0.Frame).graphic.Store(r);
    OS.Register(file)
END Store;

END Draw0.
```

12.4.4 Rechtecke als spezielle Figuren

Der bisher entwickelte Grafikeditor kann nur mit *abstrakten* Figuren arbeiten. Man kann ihn aber erweitern, indem man von den abstrakten Figuren konkrete Unterklassen für Rechtecke, Kreise oder Linien ableitet. Für jede konkrete Figurenart schreibt man ein eigenes Modul und fügt es zum bestehenden Editor hinzu (Abb. 12.28).

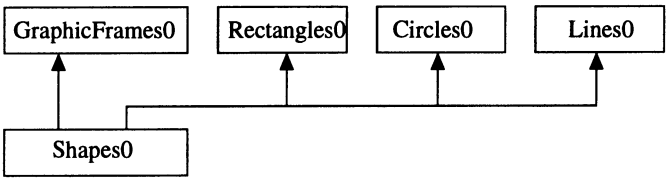


Abb. 12.28 Modulhierarchie mit Figuren-Erweiterung

Das Oberon-System erlaubt sogar, Module wie *Rectangles0* erst zur Laufzeit hinzuzuladen, während man mit dem Grafikeditor arbeitet. Dadurch kann man die Mächtigkeit des Editors dynamisch den Bedürfnissen des Benutzers anpassen.

Wir greifen als Beispiel einer Figuren-Erweiterung das Modul *Rectangles0* heraus, in dem Rechtecke implementiert sind. Seine Schnittstelle lautet:

Schnittstelle von Rectangles0

```
DEFINITION Rectangles0;
IMPORT Shapes0;

TYPE
    Rectangle = POINTER TO RectDesc;
    RectDesc = RECORD (Shapes0.ShapeDesc)
        PROCEDURE (r: Rectangle) Draw (f: Viewers0.Frame);
        PROCEDURE (r: Rectangle) Move (dx, dy: INTEGER);
        PROCEDURE (r: Rectangle) SetSelection (x, y, w, h: INTEGER);
        PROCEDURE (r: Rectangle) SetBox (x, y, w, h: INTEGER);
        PROCEDURE (r: Rectangle) GetBox (VAR x, y, w, h: INTEGER);
        PROCEDURE (r: Rectangle) Copy (): Shapes0.Shape;
        PROCEDURE (r: Rectangle) Load (VAR R: OS.Rider);
```

```
        PROCEDURE (r: Rectangle) Store (VAR R: OS.Rider);
    END;

PROCEDURE Set;

END Rectangles0.
```

Rectangle hat die gleiche Schnittstelle wie *Shapes0.Shape*. In der Implementierung von *Rectangle* werden aber die abstrakten Methoden überschrieben.

Das Kommando *Rectangles0.Set* installiert ein Rechteck als Prototyp-Objekt in der Variablen *Shapes0.curShape*, so daß in der Folge durch Drücken des linken Mausknopfs in einem Grafikrahmen Rechtecke gezeichnet werden.

Implementierung von Rectangles0

```
MODULE Rectangles0;
IMPORT OS, Viewers0, Shapes0, GraphicFrames0;

TYPE
    Rectangle* = POINTER TO RectDesc;
    RectDesc* = RECORD (Shapes0.ShapeDesc)
        x, y, w, h: INTEGER
    END;

PROCEDURE (r: Rectangle) SetBox* (x, y, w, h: INTEGER);
BEGIN
    r.SetBox^ (x, y, w, h);
    r.x := x; r.y := y; r.w := w; r.h := h
END SetBox;

PROCEDURE (r: Rectangle) Draw* (f: Viewers0.Frame);
BEGIN
    WITH f: GraphicFrames0.Frame DO
        IF r.selected THEN f.InvertBlock(r.x, r.y, r.w, r.h)
        ELSE
            f.InvertBlock(r.x, r.y, r.w, 1);
            f.InvertBlock(r.x, r.y + r.h - 1, r.w, 1);
            f.InvertBlock(r.x, r.y + 1, 1, r.h - 2);
            f.InvertBlock(r.x + r.w - 1, r.y + 1, 1, r.h - 2)
        END
    END
END Draw;

PROCEDURE (r: Rectangle) Move* (dx, dy: INTEGER);
BEGIN
    INC(r.x, dx); INC(r.y, dy)
END Move;

PROCEDURE (r: Rectangle) SetSelection* (x, y, w, h: INTEGER);
BEGIN
    r.selected := (r.x >= x) & (r.x+r.w <= x+w) & (r.y >= y) & (r.y+r.h <= y+h)
```

```
END SetSelection;

PROCEDURE (r: Rectangle) GetBox* (VAR x, y, w, h: INTEGER);
BEGIN
   x := r.x; y := r.y; w := r.w; h := r.h
END GetBox;

PROCEDURE (r: Rectangle) Copy* (): Shapes0.Shape;
   VAR r1: Rectangle;
BEGIN
   NEW(r1);
   r1.selected := r.selected; r1.x := r.x; r1.y := r.y; r1.w := r.w; r1.h := r.h;
   RETURN r1
END Copy;

PROCEDURE (r: Rectangle) Load* (VAR R: OS.Rider);
BEGIN
   R.ReadInt(r.x); R.ReadInt(r.y); R.ReadInt(r.w); R.ReadInt(r.h)
END Load;

PROCEDURE (r: Rectangle) Store* (VAR R: OS.Rider);
BEGIN
   R.WriteInt(r.x); R.WriteInt(r.y); R.WriteInt(r.w); R.WriteInt(r.h)
END Store;

PROCEDURE Set*;
   VAR r: Rectangle;
BEGIN
   NEW(r); r.SetBox(0, 0, 0, 0); r.selected := FALSE;
   Shapes0.curShape := r
END Set;

END Rectangles0.
```

Rectangles0.Set erzeugt ein neues Rechteck und legt es als Prototyp-Objekt in der globalen Variablen *Shapes0.curShape* ab. Der Grafikrahmen benutzt dieses Objekt bei der Erzeugung neuer Figuren.

12.5 Einbettung von Grafiken in Texte

Der nächste Schritt ist, Grafik in Texte zu integrieren. Wir wollen Bilder in Texte einfügen und sie beim Editieren von Texten "mitfließen" lassen. Zum Glück haben wir bereits dafür gesorgt, daß Texte unbekannte Elemente behandeln können. Bilder sind dann einfach eine besondere Art von Elementen, nämlich *Grafikelemente*.

Grafikelemente

Wie muß sich ein Grafikelement verhalten: Man erzeugt es mit einem Kommando *GraphicElems0.Insert*, wobei ein leeres Grafikelement (dargestellt durch ein leeres Rechteck) an der Caret-Position in

einen Text eingefügt wird. Klickt man es mit dem mittleren Mausknopf an, öffnet sich ein Grafikfenster, das die im Element enthaltene Zeichnung anzeigt. In diesem Fenster kann man editieren. Wenn man die Zeichnung wieder ins Element übernehmen will, klickt man das Kommando *GraphicElems0.Update* im Menü des Grafikfensters an (Abb. 12.29).

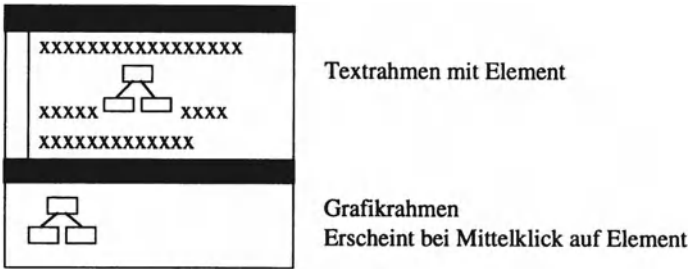


Abb. 12.29 Verhalten von Grafikelementen

Ein Grafikelement ist eine Unterklasse von *Texts0.Element*. Es enthält eine Liste von Figuren, die in einem Grafikrahmen dargestellt werden können.

Wie wird ein Grafikelement mitten in einem Textrahmen dargestellt? Dazu legt man an die Stelle, an der das Element im Textrahmen erscheinen soll, einen (temporären) Grafikrahmen mit den Ausmaßen des Elements. In diesem Rahmen kann man die Figurenliste des Elements anzeigen. Er muß nur vorhanden sein, während der Inhalt des Elements gezeichnet wird, anschließend kann er entfernt werden. Man kommt also mit einem einzigen Grafikrahmen für alle Grafikelemente aus.

Wir implementieren Grafikelemente in einem Modul namens *GraphicElems0*, das folgende Schnittstelle aufweist:

Schnittstelle von GraphicElems0

```
DEFINITION GraphicElems0;
IMPORT OS, Texts0;

TYPE
   Element = POINTER TO ElemDesc;
   ElemDesc = RECORD (Texts0.ElemDesc)
      PROCEDURE (e: Element) Draw  (x, y: INTEGER);
      PROCEDURE (e: Element) HandleMouse  (f: OS.Object;
         x, y: INTEGER);
      PROCEDURE (e: Element) Copy  (): Texts0.Element;
      PROCEDURE (e: Element) Load  (VAR r: OS.Rider);
      PROCEDURE (e: Element) Store  (VAR r: OS.Rider);
   END;

PROCEDURE Insert;
```

```
PROCEDURE Update;

END GraphicElems0.
```

Die Klasse *GraphicElems0.Element* hat die gleiche Schnittstelle wie ihre abstrakte Basisklasse *Texts0.Element*. Die geerbten Methoden werden gemäß dem gewünschten Verhalten von Grafikelementen überschrieben.

Implementierung von GraphicElems0

```
MODULE GraphicElems0;
IMPORT OS, Texts0, Shapes0, GraphicFrames0, TextFrames0, Viewers0;

TYPE
   Element* = POINTER TO ElemDesc;
   ElemDesc* = RECORD (Texts0.ElemDesc)
      orgX, orgY: INTEGER;
      graphic: Shapes0.Graphic;
   END;
   UpdateFrame = POINTER TO UpdateFrameDesc;
   UpdateFrameDesc = RECORD (GraphicFrames0.FrameDesc)
      text: Texts0.Text;
      e: Element
   END;

VAR f: GraphicFrames0.Frame;
   (*reused within a text frame whenever a graphic element has to be redrawn*)

PROCEDURE (e: Element) Copy* (): Texts0.Element;
   VAR res: Element;
BEGIN
   NEW(res); res^ := e^; res.graphic := e.graphic.Copy(); RETURN res
END Copy;

PROCEDURE (e: Element) Draw* (x, y: INTEGER);                    1 ☞
BEGIN
   f.x := x; f.y := y; f.w := e.w; f.h := e.h;
   f.orgX := e.orgX; f.orgY := e.orgY; f.graphic := e.graphic;
   f.Draw
END Draw;

PROCEDURE (e: Element) HandleMouse* (f: OS.Object; x, y: INTEGER);   2 ☞
   VAR v: Viewers0.Viewer; menu: TextFrames0.Frame;
      cont: UpdateFrame; buttons: SET;
BEGIN
   REPEAT OS.GetMouse(buttons, x, y) UNTIL buttons = {};
   menu := TextFrames0.NewMenu
      ("", "Viewers0.Close  Viewers0.Copy  GraphicElems0.Update");
   NEW(cont);
   cont.graphic := e.graphic;
   cont.orgX := e.orgX + 10; cont.orgY := e.orgY + 10;
   cont.text := f(TextFrames0.Frame).text; cont.e := e;
   v := Viewers0.New(menu, cont)
```

```
END HandleMouse;

PROCEDURE (e: Element) Load* (VAR r: OS.Rider);
BEGIN
   e.Load^ (r);
   r.ReadInt(e.orgX); r.ReadInt(e.orgY);
   NEW(e.graphic); Shapes0.InitGraphic(e.graphic); e.graphic.Load(r)
END Load;

PROCEDURE (e: Element) Store* (VAR r: OS.Rider);
BEGIN
   e.Store^ (r); r.WriteInt(e.orgX); r.WriteInt(e.orgY); e.graphic.Store(r)
END Store;

PROCEDURE Insert*;
   VAR e: Element; f: TextFrames0.Frame;
BEGIN
   IF Viewers0.focus # NIL THEN
      f := Viewers0.focus(TextFrames0.Frame);
      IF (f # NIL) & (f.caret.pos >= 0) THEN
         NEW(e); e.w := 12; e.h := 12; e.dsc := 0;
         NEW(e.graphic); Shapes0.InitGraphic(e.graphic);
         e.orgX := 0; e.orgY := 0;
         f.text.SetPos(f.caret.pos); f.text.WriteElem(e)
      END
   END
END Insert;

3 ☞ PROCEDURE Update*;
   VAR v: Viewers0.Viewer; f: UpdateFrame; e: Element;
      m: Texts0.NotifyReplMsg; x, y: INTEGER; pos: LONGINT;
BEGIN
   v := Viewers0.ViewerAt(TextFrames0.cmdFrame.y);
   f := v.cont(UpdateFrame);
   e := f.e; pos := f.text.ElemPos(e);
   IF pos < f.text.len THEN
      f.graphic.GetBox(x, y, e.w, e.h);
      e.graphic := f.graphic; e.orgX := - x ; e.orgY := - y;
      m.t := f.text; m.beg := pos; m.end := pos + 1; Viewers0.Broadcast(m)
   END
END Update;

PROCEDURE Init;
   VAR g: Shapes0.Graphic;
BEGIN
   NEW(g); Shapes0.InitGraphic(g); f := GraphicFrames0.New(g)
END Init;

BEGIN
   Init
END GraphicElems0.
```

Draw soll das Grafikelement an der Position *(x, y)* am Bildschirm zeichnen. Dazu legt es einen (temporären) Grafikrahmen mit der nötigen Größe an diese Position, installiert die Figurenliste in ihm und schickt ihm eine *Draw*-Meldung. 1 ☞

HandleMouse wird aufgerufen, wenn ein Grafikelement mit dem mittleren Mausknopf angeklickt wird. Die Prozedur öffnet ein Grafikfenster mit einem Grafikrahmen vom Typ *UdateFrame* und zeigt darin die Figuren des Elements an. Das *UpdateFrame*-Objekt enthält einen Hinweis darauf, welches Element in ihm gerade editiert wird (*f.e*) und aus welchem Text es stammt (*f.text*). Diese Informationen sind nötig, um später im *Update*-Kommando die editierte Figurenliste wieder ins Element zurückspeichern zu können. 2 ☞

In einem *UpdateFrame*-Objekt *f* wird die Figurenliste des Elements *f.e* editiert, das aus dem Text *f.text* stammt. *Update* schreibt die editierte Figurenliste nach *f.e* zurück. Die Größe von *f.e* wird als das kleinste Rechteck bestimmt, das alle Figuren umschließt. 3 ☞

Was kann man aus dieser Implementierung lernen?

Dieses Beispiel zeigt, wie in Oberon zwei ursprünglich nicht aufeinander abgestimmte Programme integriert werden können. Folgende Eigenschaften sind dafür maßgeblich:

1. *GraphicElems0* hat Zugriff auf den Text im Texteditor, was nötig ist, um darin Elemente einfügen zu können. Er hat auch Zugriff auf die Figuren in einem Grafikrahmen, was nötig ist, um die Figuren eines Elements in einem Grafikrahmen anzuzeigen. Oberon-Programme sind also keine abgeschlossenen Monolithe, sondern sie sind offen in dem Sinne, daß exportierte Datenstrukturen für andere Programme sichtbar sind.

2. Grafikelemente sind kompatibel mit abstrakten Elementen und können daher vom Texteditor wie Elemente behandelt werden. Der Editor kommuniziert mit ihnen über Meldungen und nicht über Prozeduraufrufe. Meldungen stellen eine losere Kopplung von Programmteilen dar als Prozeduren, die voraussetzen, daß der Rufer die gerufene Prozedur kennt.

Grafikelemente sind ein Beispiel für das in Kapitel 9.3.2 beschriebene *Adapter*-Muster. Sie machen eine Figurenmenge zu Textelementen kompatibel.

13 Kosten und Nutzen

Das folgende Kapitel faßt die Aussagen dieses Buches zusammen: Warum sollte jemand objektorientiert programmieren statt prozedural? Was ist der Nutzen der objektorientierten Programmierung und was sind ihre Kosten? Ist der Nutzen größer als die Kosten?

Wenn man die Stärken und Schwächen der objektorientierten Programmierung kennt und Klassen sinnvoll einsetzt, ist ihr Nutzen bei weitem größer als ihre Kosten. Die Kosten können aber rasch ansteigen, wenn man Klassen unkritisch verwendet, und zwar in Situationen, in denen sie keine Erleichterung bringen, sondern nur zusätzliche Komplexität.

13.1 Nutzen

Meisterung der Komplexität

Von einer Programmiertechnik erwarten wir, daß sie uns bei der Lösung von Problemen hilft. Das größte Problem bei der Programmierung ist die *Komplexität*. Je größer und komplexer ein Programm ist, desto wichtiger ist es, daß man es in kleine, überschaubare Teile zerlegen kann. Um die Komplexität zu beherrschen, muß man von Details abstrahieren. Klassen sind dazu ein geeignetes Konstrukt:

- Klassen erlauben die Herstellung überschaubarer Bausteine mit einer einfachen Schnittstelle, die Implementierungsdetails von Klienten fernhält.
- Daten und Operationen bilden eine Einheit und sind nicht, wie es sonst oft vorkommt, weit über ein Programm verstreut.
- Die bessere Lokalität von Code und Daten fördert die Lesbarkeit und Wartbarkeit von Programmen.
- Die Datenkapselung schützt vor unberechtigtem Zugriff auf kritische Daten.

Erweiterbarkeit

Objektorientierte Programmierung ermöglicht es, erweiterbare Systeme zu bauen. Das ist ihr größter Vorteil und hebt sie von herkömmlichen,

H. Mössenböck, *Objektorientierte Programmierung in Oberon-2*
© Springer-Verlag Berlin Heidelberg 1998

prozeduralen Techniken ab. Erweiterbarkeit bedeutet, daß man bestehende Systeme ohne sie zu ändern dazu bringen kann, mit neuen Bausteinen zu arbeiten, ja daß man sogar zur Laufzeit einen alten Baustein gegen einen neuen austauschen kann.

Die Typerweiterung und der damit verbundene Polymorphismus von Variablen bringt vor allem in folgenden Situationen Vorteile (siehe Kapitel 8):

- *Verwaltung heterogener Datenstrukturen.* Programme können mit Varianten von Objekten arbeiten, ohne zwischen ihnen zu unterscheiden. Es können jederzeit neue Varianten hinzukommen.
- *Auswechseln von Verhalten zur Laufzeit.* Ein Objekt kann zur Laufzeit durch ein anderes ersetzt werden. Dadurch kann man das Verhalten eines Algorithmus ändern, der dieses Objekt benutzt.
- *Implementierung generischer Bausteine.* Algorithmen und Klassen können verallgemeinert werden, so daß sie nicht nur mit *einer* Art von Objekten arbeiten können, sondern mit verschiedenen Objektarten.
- *Ausbau von Halbfabrikaten.* Bausteine müssen nicht auf einen bestimmten Zweck zugeschnitten sein. Man kann sie als universelle Halbfabrikate in einer Bibliothek ablegen und sie bei Bedarf zu verschiedenen Endfabrikaten erweitern.
- *Erweiterung von Frameworks.* Die anwendungsunabhängigen Teile eines Aufgabenbereichs können als Framework implementiert und später durch anwendungsabhängige Teile erweitert werden.

Wiederverwendung

Die Wiederverwendung von Software scheitert in der Praxis oft daran, daß bestehende Bausteine nicht für den neuen Zweck passen. Objektorientierte Programmierung bietet die Möglichkeit, Bausteine zu erweitern und anzupassen, ohne jedoch die bisherigen Klienten der Bausteine zu invalidieren. Damit genießt man alle Vorteile der Wiederverwendung:

- Man spart Entwicklungszeit, die man anderswo nutzbringender einsetzen kann.
- Wiederverwendete Bausteine enthalten meist weniger Fehler als neu entwickelte, da sie schon mehrfach getestet wurden.
- Wenn ein Baustein mehrfach verwendet wird, wirken sich Verbesserungen an ihm gleich in mehreren Programmen aus.
- Wenn verschiedene Programme immer wieder auf Standardbausteine zurückgreifen, werden sie in ihrem Aufbau und in ihrer Benutzeroberfläche ähnlicher. Es wird einfacher, solche Programme zu verstehen und zu bedienen.

13.2 Kosten

Lernaufwand

Um objektorientiert programmieren zu können, muß man vier Dinge lernen:

1. Man muß Klassen, Vererbung und dynamische Bindung verstehen. Für Programmierer, die bereits mit Modulen und abstrakten Datentypen vertraut sind, ist das nur ein kleiner Schritt. Für andere, die Datenkapselung noch nie benutzt haben, bedeutet es einen Standortwechsel und kann einige Zeit in Anspruch nehmen.

2. Der Wunsch nach Wiederverwendung erfordert, daß man sich mit großen *Klassenbibliotheken* vertraut macht. Das ist meist viel schwieriger, als eine bestimmte Programmiersprache zu erlernen. Eine Klassenbibliothek ist ja nichts anderes als eine virtuelle Sprache, die oft Hunderte von Typen und Tausende von Operationen enthält. Man muß oft einen beträchtlichen Teil dieser Klassenbibliothek kennen, bevor man einigermaßen sinnvolle Programme schreiben kann. Das kostet Zeit.

3. Schwieriger als eine Klassenbibliothek zu *benutzen*, ist es, eine neue zu *entwerfen*. Klassenentwurf ist *Sprachentwurf* und erfordert Erfahrung. Es ist ein iterativer Prozeß, bei dem man durch Fehler lernt.

4. Ebenso schwierig ist es, zu lernen, in welchen Situationen Klassen am Platz sind (siehe Kapitel 8 und 9) und in welchen sie nichts bringen, ja vielleicht sogar eher Kosten verursachen. Erst wenn man diese kritische Fähigkeit hat, beherrscht man die objektorientierte Programmierung.

Der Lernaufwand für die Grundkonzepte ist also gering, der für die Klassenbibliothek und für den richtigen Einsatz von Klassen jedoch groß.

Verständnisprobleme

Da man eine Bibliotheksklasse meist nicht im Quellcode vor sich hat, ist man darauf angewiesen, sie allein aufgrund ihrer Dokumentation und der Namensgebung zu verstehen. Die Zeit, die man auf der einen Seite gewinnt, indem man die Klasse nicht selbst schreiben muß, muß man also zum Teil (besonders am Anfang) wieder investieren, um die Klasse zu verstehen.

Die Dokumentation von Klassen ist schwieriger als die von Prozeduren oder Modulen. Da jede Methode überschrieben werden kann, muß

man nicht nur dokumentieren, was sie selbst leistet, sondern auch, was überschreibende Methoden leisten oder zumindest beachten sollen. Das gilt vor allem für abstrakte Methoden, die ja selbst noch nichts leisten. Hier muß man auf jeden Fall angeben, was man von der überschreibenden Methode erwartet. Insbesondere ist es wichtig zu sagen, in welchem Zusammenhang eine Methode aufgerufen wird: oft ruft man überschriebene Methoden nicht selbst auf, sondern sie werden von einem Framework aufgerufen. Man muß dann als Implementierer wissen, was zu diesem Zeitpunkt gilt.

Bei tiefen Klassenhierarchien sind die Attribute und Methoden einer Klasse meist über mehrere Hierarchieebenen verteilt. Es ist nicht immer einfach zu sehen, welche Attribute oder Methoden eine Klasse nun eigentlich hat. Man braucht Werkzeuge, wie einen *Browser*, die einem diese Information liefern. Werden konkrete Klassen erweitert, dann leistet jede Methode nur wenig und delegiert den Rest der Aufgabe an die Basisklasse. Die Implementierung einer Operation ist so auf mehrere Klassen verteilt, und man muß manchmal lange blättern, bis man ihre Wirkung versteht. Wenn man beim Studium einer Methode ständig in der Klassenhierarchie auf und absteigen muß, nennt man das den *Jojo-Effekt*.

Methoden sind meist kürzer als Prozeduren, da sie nur eine einzige Operation auf Daten ausführen. Dafür ist ihre Anzahl umso größer. Kurze Methoden haben den Vorteil, daß sie einfach zu verstehen sind, aber den Nachteil, daß der Ablauf des gesamten Programms auf viele kleine Methoden verteilt ist.

Flexibilität

Datenabstraktion schränkt die Flexibilität der Klienten ein. Klienten können mit einem Objekt nur noch jene Operationen ausführen, die seine Klasse anbietet. Sie können nicht mehr beliebig auf seine Daten zugreifen. Das ist aber meist gewollt, denn schließlich verwendet man ja aus dem gleichen Grund höhere Programmiersprachen, damit gewisse unsaubere Programmstrukturen nicht mehr möglich sind.

Datenabstraktion soll man nicht übertreiben. Je mehr Daten man versteckt, desto schwieriger wird es, eine Klasse zu erweitern. Es geht weniger darum, daß Klienten Daten nicht kennen *dürfen*, sondern vielmehr darum, daß sie sie nicht kennen *müssen*, um mit einem Baustein zu arbeiten.

Effizienz

Man hört oft das Argument, objektorientierte Programmierung sei ineffizient. Was ist daran wahr? Man muß zwischen Laufzeiteffizienz, Speicherbedarf und unnötiger Allgemeinheit unterscheiden:

1. *Laufzeiteffizienz*. In Sprachen wie Smalltalk werden Meldungen zur Laufzeit interpretiert, indem die dazugehörige Methode in ei-

ner Tabelle gesucht wird. Das ist natürlich langsam. Smalltalk-Programme sind daher selbst mit den besten Optimierungstechniken etwa fünf bis zehnmal so langsam wie optimierte C-Programme [Cha92].

In hybriden Sprachen wie Oberon-2, Java oder C++ kostet die Interpretation einer Meldung nur einen Zeiger-Zugriff und einen Prozeduraufruf. Auf gewissen Maschinen sind Meldungen weniger als 10% langsamer als Prozeduraufrufe. Da Meldungen im Vergleich zu anderen Operationen selten sind, ist ihr Einfluß auf die Gesamtlaufzeit kaum spürbar.

Es gibt allerdings noch einen anderen Faktor, der die Laufzeit beeinflußt: die Datenabstraktion. Sie bedingt, daß auf die Attribute eines Objekts nicht mehr direkt zugegriffen wird, sondern über Methoden. Das kostet einen zusätzlichen Prozeduraufruf bei jedem Datenzugriff. Wenn man aber mit Klassen und Datenabstraktion vernünftig umgeht, sind die Auswirkungen auf die Laufzeit eines Programms gering.

2. *Speicherbedarf.* Für die dynamische Bindung und den Typtest braucht man zur Laufzeit Informationen über den Typ jedes Objekts. Pro Klasse existiert ein *Typdeskriptor*, der diese Informationen enthält. Jedes Objekt hat einen (für den Programmierer unsichtbaren) Zeiger auf den Typdeskriptor seiner Klasse. Der zusätzliche Speicherbedarf in objektorientierten Programmen ist also ein Zeiger pro Objekt und ein Typdeskriptor pro Klasse.

3. *Unnötige Allgemeinheit.* Ineffizienz kann auch bedeuten, daß ein Programm Eigenschaften besitzt, die nicht benutzt werden. Eine Bibliotheksklasse hat meist viel mehr Methoden als man braucht. Da man überflüssige Methoden nicht entfernen kann, muß man sie mitschleppen, auch wenn man sie nicht benutzt. Das hat zwar keinen Einfluß auf die Laufzeit, wohl aber auf die Codegröße.

   Ein möglicher Ausweg besteht darin, eine Basisklasse mit einem Minimum an Methoden anzubieten und dann verschiedene Erweiterungen davon zu implementieren, die mehr und mehr Funktionalität hinzufügen.

   Ein anderer Ausweg ist, überflüssige Methoden durch den Binder entfernen zu lassen. Solche sogenannten "*Smart Linkers*" sind für verschiedene Sprachen und Betriebsysteme vorhanden.

   Oberon besitzt eine andere Art, unnötige Allgemeinheit zu vermeiden: Programmteile können zur Laufzeit hinzugefügt werden. Damit muß nicht immer das gesamte Programm geladen werden, sondern nur derjenige Teil, den der Benutzer gerade

braucht. Das spart in der Praxis viel mehr Code als das Entfernen einzelner Methoden.

Es ist also im allgemeinen nicht wahr, daß objektorientierte Programmierung ineffizient ist. Wenn man Klassen gezielt nur dort einsetzt, wo sie sinnvoll sind, ist die Effizienzeinbuße weder in der Laufzeit noch im Speicherbedarf spürbar.

13.3 Ausblick

Wird sich die objektorientierte Programmierung durchsetzen oder ist sie nur eine Modewelle, die sich bald wieder legt?

Klassen finden derzeit Eingang in fast alle modernen Programmiersprachen. Das deutet schon darauf hin, daß sie bleiben werden. Ihr Einsatz wird bald zum selbstverständlichen Repertoire jedes Programmierers gehören, so wie heute jeder Programmierer den Umgang mit dynamischen Datenstrukturen oder rekursiven Prozeduren beherrscht, die ja auch einmal neu waren. Klassen sind einfach *ein* neues Konstrukt neben vielen anderen. Man muß lernen, für welche Situationen sie sich eignen und wird sie dann dort und nur dort verwenden. Es gehört zu den Fähigkeiten jedes Handwerkers und erst recht jedes Ingenieurs, für jede Aufgabe das richtige Werkzeug zu wählen.

Es herrscht zur Zeit eine gewisse Euphorie bezüglich der objektorientierten Programmierung. Zeitschrifteninserate versprechen Unglaubliches und selbst manche Forscher scheinen die objektorientierte Programmierung für eine Wunderwaffe zu halten, die alle Probleme der Softwareentwicklung beseitigt. Diese Euphorie wird sich legen. Nach einer Zeit der Ernüchterung wird man vielleicht nicht mehr ausdrücklich von objektorientierter Programmierung sprechen, so wie man heute auch kaum noch von strukturierter Programmierung spricht. Man wird aber Klassen ganz selbstverständlich verwenden und als das sehen, was sie sind: Bausteine, die helfen, modulare und erweiterbare Software zu entwickeln.

A Sprachdefinition Oberon-2

A.1 Einleitung

Oberon-2 ist eine universelle Programmiersprache in der Tradition von Pascal und Modula-2. Sie bietet getrennt übersetzbare Module, strenge Typenprüfung auch über Modulgrenzen hinweg sowie Objektorientiertheit.

Die Sprachdefinition ist bewußt knapp gehalten. Sie soll kein Programmierlehrbuch sein, sondern Programmierern, Übersetzerbauern und Handbuch-Autoren als Referenz dienen. Wenn etwas undefiniert bleibt, geschieht das meist, weil es aus den angegebenen Regeln der Sprache hervorgeht oder weil es die Sprachdefinition unnötig einengen würde.

Abschnitt A.12.1 definiert einige Begriffe, die zur Beschreibung der Kontextbedingungen von Oberon-2 benötigt werden. Wo sie im Text vorkommen, werden sie kursiv geschrieben, um ihre besondere Bedeutung hervorzuheben (z.B. *derselbe* Typ).

A.2 Syntax

Die Syntax von Oberon-2 wird in erweiterter *Backus-Naur-Form* (*EBNF*) beschrieben: Alternativen werden durch | getrennt. Eckige Klammern [und] umschließen Ausdrücke die fehlen dürfen, geschweifte Klammern { und } umschließen (null- oder mehrmals) wiederholbare Ausdrücke. Nonterminalsymbole beginnen mit einem Großbuchstaben (z.B. Statement). Terminalsymbole beginnen entweder mit einem Kleinbuchstaben (z.B. letter) oder werden ganz mit Großbuchstaben geschrieben (z.B. BEGIN) oder sind Zeichenketten (z.B. ":=").

A.3 Terminalsymbole

Terminalsymbole werden im ASCII-Zeichensatz dargestellt, wobei Groß- und Kleinbuchstaben als verschieden betrachtet werden. Als Terminalsymbole gelten Namen, Zahlen, Zeichenketten, Operatoren und Begrenzer. In ihnen dürfen keine Leerzeichen und Zeilenumbrüche vorkommen (außer Leerzeichen in Zeichenketten). Leerzeichen und Zeilenumbrüche dienen zur Trennung von Symbolen und haben keine andere Bedeutung.

1. *Namen* sind Folgen von Buchstaben oder Ziffern beginnend mit einem Buchstaben.

   ```
   ident = letter {letter | digit}.
   ```

   Beispiele: *x Scan Oberon2 GetSymbol firstLetter*

2. *Zahlen* sind (vorzeichenlose) ganzzahlige oder reelle Konstanten. Der Typ einer ganzzahligen Konstanten ist der kleinste Typ, der den Konstantenwert einschließt (siehe A.6.1). Endet eine ganzzahlige Konstante mit dem Buchstaben H, so ist ihre Darstellung hexadezimal, sonst dezimal.

   Eine reelle Zahl enthält immer einen Dezimalpunkt und wahlweise einen dezimalen Exponenten. Der Buchstabe E (oder D) bedeutet "mal zehn hoch". Enthält eine reelle Zahl den Buchstaben E im Exponenten, ist ihr Typ REAL, enthält sie den Buchstaben D im Exponenten, ist ihr Typ LONGREAL.

   ```
   number = integer | real.
   integer = digit {digit} | digit {hexDigit} "H".
   real = digit {digit} "." {digit} [ScaleFactor].
   ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
   hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
   digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
   ```

   Beispiele:

Zahl	*Typ*	*Wert*
1991	INTEGER	1991
0DH	SHORTINT	13
12.3	REAL	12.3
4.567E8	REAL	456700000
0.57712566D-6	LONGREAL	0.00000057712566

3. *Zeichenkonstanten* werden durch ihren numerischen Wert angegeben, gefolgt vom Buchstaben X.

character = digit {hexDigit} "X".

4. *Zeichenketten* sind Zeichenfolgen zwischen einfachen (') oder doppelten (") Hochkommas. Das schließende Hochkomma muß gleich sein wie das öffnende und darf in der Zeichenkette nicht vorkommen. Die *Länge* einer Zeichenkette ist die Anzahl der in ihr enthaltenen Zeichen. Eine Zeichenkette der Länge 1 kann überall dort benutzt werden, wo eine Zeichenkonstante erlaubt ist und umgekehrt.

   string = '"' {char} '"' | "'" {char} "'".

   Beispiele: "Oberon-2" "Don't worry" "x"

5. Als *Operatoren* und *Begrenzer* gelten die folgenden Sonderzeichen, Zeichenpaare und Schlüsselwörter. Schlüsselwörter bestehen ausschließlich aus Großbuchstaben und dürfen nicht als Namen verwendet werden.

+	:=	ARRAY	IMPORT	RETURN
-	^	BEGIN	IN	THEN
*	=	BY	IS	TO
/	#	CASE	LOOP	TYPE
~	<	CONST	MOD	UNTIL
&	>	DIV	MODULE	VAR
.	<=	DO	NIL	WHILE
,	>=	ELSE	OF	WITH
;	..	ELSIF	OR	
|	:	END	POINTER	
(	)	EXIT	PROCEDURE	
{	}	IF	REPEAT	

6. *Kommentare* sind beliebige Zeichenfolgen zwischen den Klammern (* und *). Sie können überall zwischen Symbolen vorkommen und haben keinen Einfluß auf die Bedeutung eines Programms. Kommentare dürfen geschachtelt werden.

A.4 Deklarationen, Sichtbarkeitsbereiche

Jeder in einem Programm vorkommende Name muß durch eine *Deklaration* eingeführt werden (mit Ausnahme von vordeklarierten Namen). Die Deklaration legt auch gewisse bleibende Eigenschaften eines Objekts fest, zum Beispiel ob es eine Konstante, ein Typ, eine Variable oder eine Prozedur ist. Über den Namen kann man sich später auf das betreffende Objekt beziehen.

Der *Sichtbarkeitsbereich* eines Namens *x* erstreckt sich textuell von seiner Deklaration bis zum Ende des (Modul-, Prozedur- oder Record-) *Blocks*, zu dem die Deklaration gehört und zu dem das benannte Objekt daher *lokal* ist. Er schließt die Sichtbarkeitsbereiche gleicher Namen aus, die in geschachtelten Blöcken deklariert werden. Es gelten die folgenden Regeln:

1. Kein Name darf innerhalb seines Sichtbarkeitsbereiches mehr als ein Objekt bezeichnen (d.h. kein Name darf in einem Block mehr als einmal deklariert werden);
2. Ein Name darf nur innerhalb seines Sichtbarkeitsbereichs benutzt werden;
3. Ein Typ *T* der Form POINTER TO *T1* (siehe A.6.4) kann textuell vor *T1* deklariert werden. Die Deklaration von *T1* muß aber im gleichen Block erfolgen, zu dem *T* lokal ist;
4. Namen von Recordfeldern (siehe A.6.3) oder typgebundenen Prozeduren (siehe A.10.2) dürfen nur innerhalb von Record-Bezeichnern verwendet werden.

Wenn ein Name bei seiner Deklaration in einem Modul-Block von einer *Export-Marke* ("*" oder "-") gefolgt wird, bedeutet dies, daß er exportiert wird. Ein von einem Modul *M* exportierter Name *x* darf in anderen Modulen verwendet werden, die *M* importieren (siehe A.11). Er wird in diesen Modulen als *M.x* geschrieben und heißt *qualifizierter Name*. Variablen und Recordfelder, die bei ihrer Deklaration mit "-" markiert werden, sind in importierenden Modulen schreibgeschützt.

```
Qualident = [ident "."] ident.
IdentDef = ident ["*" | "-"].
```

Die folgenden Namen sind vordeklariert; ihre Bedeutung wird in den angegebenen Abschnitten beschrieben:

ABS	(A.10.3)	FALSE	(A.6.1)	NEW	(A.10.3)
ASH	(A.10.3)	HALT	(A.10.3)	ODD	(A.10.3)
ASSERT	(A.10.3)	INC	(A.10.3)	ORD	(A.10.3)
BOOLEAN	(A.6.1)	INCL	(A.10.3)	REAL	(A.6.1)
CAP	(A.10.3)	INTEGER	(A.6.1)	SET	(A.6.1)
CHAR	(A.6.1)	LEN	(A.10.3)	SHORT	(A.10.3)
CHR	(A.10.3)	LONG	(A.10.3)	SHORTINT	(A.6.1)
COPY	(A.10.3)	LONGINT	(A.6.1)	SIZE	(A.10.3)
DEC	(A.10.3)	LONGREAL	(A.6.1)	TRUE	(A.6.1)
ENTIER	(A.10.3)	MAX	(A.10.3)		
EXCL	(A.10.3)	MIN	(A.10.3)		

A.5 Konstantendeklarationen

Eine Konstantendeklaration verknüpft einen Namen mit einem Konstantenwert.

```
ConstantDeclaration = IdentDef "=" ConstExpression.
ConstExpression = Expression.
```

Ein Konstantenausdruck ist ein Ausdruck, der ausgewertet werden kann, ohne das Programm auszuführen. Seine Operanden sind Konstanten (A.8) oder vordeklarierte Funktionen (A.10.3), deren Wert zur Übersetzungszeit bekannt ist. Beispiele für Konstantendeklarationen sind:

```
N = 100
limit = 2*N - 1
fullSet = {MIN(SET) .. MAX(SET)}
```

A.6 Typdeklarationen

Ein Datentyp bestimmt die Menge der Werte, die Variablen dieses Typs annehmen können, sowie die Operatoren, die auf diesen Typ anwendbar sind. Eine Typdeklaration verknüpft einen Namen mit einem Datentyp. Bei strukturierten Typen (Arrays und Records) definiert sie auch die Struktur von Variablen dieses Typs.

```
TypeDeclaration = IdentDef "=" Type.
Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.
```

Beispiele:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node = RECORD
    key: INTEGER;
    left, right: Tree
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
    width: INTEGER;
    subnode: Tree
END
Function = PROCEDURE (x: INTEGER): INTEGER
```

A.6.1 Standardtypen

Standardtypen werden durch vordeklarierte Namen bezeichnet. Die auf einen Standardtyp anwendbaren Operatoren sind in Abschnitt A.8.2 beschrieben, die Funktionen MIN und MAX in Abschnitt A.10.3. Die Standardtypen haben folgende Wertebereiche:

1.	BOOLEAN	die Wahrheitswerte TRUE und FALSE
2.	CHAR	die Zeichen des erweiterten ASCII-Zeichensatzes (0X..0FFX)
3.	SHORTINT	die ganzen Zahlen zwischen MIN(SHORTINT) und MAX(SHORTINT)
4.	INTEGER	die ganzen Zahlen zwischen MIN(INTEGER) und MAX(INTEGER)
5.	LONGINT	die ganzen Zahlen zwischen MIN(LONGINT) und MAX(LONGINT)
6.	REAL	die reellen Zahlen zwischen MIN(REAL) und MAX(REAL)
7.	LONGREAL	die reellen Zahlen zwischen MIN(LONGREAL) und MAX(LONGREAL)
8.	SET	die Mengen der ganzen Zahlen zwischen MIN(SET) und MAX(SET)

Die Typen 3 bis 5 werden *ganzzahlige* Typen genannt, die Typen 6 und 7 *reelle* Typen; zusammen heißen sie *numerische* Typen. Sie bilden eine Hierarchie: der größere Typ *schließt* den kleineren Typ (d.h. seine Werte) *ein*.

LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

A.6.2 Arraytypen

Ein Array ist eine Folge von Elementen gleichen Typs, den man den *Elementtyp* nennt. Die *Länge* eines Arrays ist die Anzahl seiner Elemente. Die Elemente werden durch Indizes bezeichnet, die ganze Zahlen im Bereich 0 bis Arraylänge - 1 sind.

```
ArrayType = ARRAY [Length {"," Length}] OF Type.
Length = ConstExpression.
```

Ein Typ der Form

```
ARRAY L0, L1, ..., Ln OF T
```

ist eine Kurzform für

```
ARRAY L0 OF
   ARRAY L1 OF
      ...
         ARRAY Ln OF T
```

Arrays, die ohne Längenangabe deklariert werden, nennt man *offene Arrays*. Sie dürfen nur als Zeiger-Basistypen (siehe A.6.4), als Typen formaler Parameter (siehe A.10.1) und als Elementtypen anderer offener Arrays verwendet werden.

Beispiele:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

A.6.3 Recordtypen

Ein Recordtyp besteht aus einer festen Anzahl von *Feldern* mit möglicherweise verschiedenen Typen. Die Deklaration eines Recordtyps definiert den Namen und den Typ jedes Feldes. Der Sichtbarkeitsbereich der Feldnamen erstreckt sich von ihrer Deklaration bis zum Ende der Record-Deklaration; sie sind aber auch in Bezeichnern sichtbar, die Felder von Recordvariablen ansprechen (siehe A.8.1). Wird ein Recordtyp exportiert, müssen Felder, die außerhalb des deklarierenden Moduls sichtbar sein sollen, mit einer Exportmarke versehen werden. Diese Felder sind *öffentlich*; unmarkierte Felder sind *privat*.

```
RecordType = RECORD ["(" BaseType ")"] FieldList {";" FieldList} END.
BaseType = Qualident.
FieldList = [IdentList ":" Type].
```

Ein Recordtyp kann als *Erweiterung* eines anderen deklariert werden:

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 ist eine (direkte) Erweiterung von *T0*; *T0* ist der (direkte) Basistyp von *T1* (siehe A.12.1). Ein erweiterter Typ *T1* besteht aus den Feldern seines Basistyps *T0* und den in *T1* deklarierten Feldern. Die in der Erweiterung und in den Basistypen deklarierten Namen müssen verschieden sein. Beispiele für Record-Deklarationen sind:

```
RECORD
   day, month, year: INTEGER
END

RECORD
   name, firstname: ARRAY 32 OF CHAR;
   age: INTEGER;
   salary: REAL
END
```

A.6.4 Zeigertypen

Variablen eines Zeigertyps *P* verweisen auf (namenlose) Variablen eines anderen Typs *T*, den man den *Zeiger-Basistyp* von *P* nennt; *T* muß ein Record- oder Arraytyp sein.

```
PointerType = POINTER TO Type.
```

Die Erweiterungsbeziehung zwischen Recordtypen wird auch auf Zeigertypen übertragen: Wenn *P* vom Typ POINTER TO *T*, *P1* vom Typ POINTER TO *T1* und *T1* eine Erweiterung von *T* ist, dann ist auch *P1* eine Erweiterung von *P*.

Wenn eine Variable *p* vom Typ POINTER TO *T* ist, bewirkt der Aufruf der vordeklarierten Prozedur NEW(*p*) (siehe A.10.3), daß eine Variable *p*^ vom Typ *T* im Freispeicher angelegt wird und *p* auf diese Variable verweist. Falls *T* ein *n*-dimensionales offenes Array ist, muß der Aufruf NEW(p, e_0, ..., e_{n-1}) lauten, wobei die *Ausdrücke* e_0, ..., e_{n-1} die gewünschten Längen des Arrays angeben.

Jede Zeigervariable kann den Wert NIL annehmen, der auf keine Variable verweist. Alle Zeigervariablen werden mit NIL initialisiert.

A.6.5 Prozedurtypen

Variablen eines Prozedurtyps *T* enthalten als Wert eine Prozedur oder NIL. Eine Prozedur *P* kann einer Variablen des Typs *T* zugewiesen werden, wenn die formalen Parameterlisten (siehe A.10.1) von *T* und *P* *übereinstimmen* (siehe A.12.1). *P* darf keine vordeklarierte oder typgebundene Prozedur sein und darf auch nicht lokal zu einer anderen Prozedur deklariert werden.

```
ProcedureType = PROCEDURE [FormalParameters].
```

A.7 Variablendeklarationen

Variablendeklarationen legen den Namen und den Typ von Variablen fest.

```
VariableDeclaration = IdentList ":" Type.
```

Record- und Zeigervariablen haben sowohl einen *statischen Typ* (den Typ, mit dem sie deklariert sind – in der Folge einfach Typ genannt) und einen *dynamischen Typ* (den Typ ihres Wertes zur Laufzeit). Bei Zeigern und Var-Parameter-Records kann der dynamische Typ eine Erweiterung des statischen Typs sein. Der statische Typ bestimmt, welche Felder und typgebundenen Prozeduren eines Records ansprechbar sind. Der dynamische Typ wird benutzt, um typgebundene Prozeduren aufzurufen (siehe A.10.2). Beispiele für Variablendeklarationen (Typen siehe A.6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
f: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
   name: ARRAY 32 OF CHAR;
   count: INTEGER
END
t, c: Tree
```

A.8 Ausdrücke

Ausdrücke sind Berechnungsregeln, in denen aus Konstanten- und Variablenwerten mittels Operatoren und Funktionsaufrufen neue Werte berechnet werden. Operanden und Operatoren können mit Klammern gruppiert werden.

A.8.1 Operanden

Mit Ausnahme von Literalen (Zahlen, Zeichenkonstanten oder Zeichenketten) und Mengenkonstruktoren werden Operanden durch *Bezeichner* ausgedrückt. Ein Bezeichner (designator) besteht aus einem qualifizierten Namen (siehe A.4) für eine Konstante, Variable oder

Prozedur, möglicherweise gefolgt von Selektoren, falls das bezeichnete Objekt ein Element eines strukturierten Typs ist.

```
Designator = Qualident {"." ident | "[" ExpressionList "]" | "(" Qualident ")"}.
ExpressionList = Expression {"," Expression}.
```

Wenn *a* ein Array ist, bezeichnet *a[e]* dasjenige Element von *a*, dessen Index durch den Wert des Ausdrucks *e* gegeben ist. Der Typ von *e* muß *ganzzahlig* sein. Ein Bezeichner der Form $a[e_0, \ldots, e_n]$ steht für $a[e_0]\ldots[e_n]$.

Wenn *r* ein Record ist, bezeichnet *r.f* das Feld *f* von *r* oder den Aufruf der typgebundenen Prozedur *f*, die zum dynamischen Typ von *r* gehört (siehe A.10.2).

Wenn *p* ein Zeiger ist, bezeichnet *p^* die Variable auf die *p* verweist. Die Bezeichner *p^.f* und *p^[e]* können abgekürzt werden zu *p.f* und *p[e]*.

Wenn *a* oder *r* schreibgeschützt sind, dann sind es auch *a[e]* und *r.f*.

Eine *Typzusicherung v(T)* garantiert, daß *v* vom dynamischen Typ *T* (oder einer Erweiterung davon) ist; ist das nicht der Fall, wird das Programm abgebrochen. Innerhalb des Bezeichners wird *v* so betrachtet, als ob *v* vom statischen Typ *T* wäre. Die Zusicherung ist anwendbar, wenn

1. *v* ein Zeiger oder ein Var-Parameter mit Recordtyp ist und
2. *T* eine Erweiterung des statischen Typs von *v* ist.

Steht ein Bezeichner für eine Konstante oder Variable, bedeutet er ihren Wert; steht er für eine Funktionsprozedur, bedeutet er diese Prozedur, außer er wird von einer (möglicherweise leeren) Parameterliste gefolgt; in diesem Fall wird die Funktionsprozedur ausgeführt und der Bezeichner steht für den Wert, den sie liefert. Die aktuellen Parameter müssen zu den formalen Parametern passen (siehe A.10.1). Beispiele für Bezeichner (siehe Beispiele in A.7):

i	INTEGER
a[i]	REAL
w[3].name[i]	CHAR
t(CenterNode).subnode	Tree

A.8.2 Operatoren

Es gibt vier Klassen von Operatoren mit verschiedenen Vorrangregeln, die in Ausdrücken syntaktisch unterschieden werden. Der Operator ~ hat die höchste Priorität, gefolgt von Multiplikations-, Additions- und

Vergleichsoperatoren. Operatoren derselben Priorität werden von links nach rechts ausgeführt: *x-y-z* steht für *(x-y)-z*.

```
Expression        =  SimpleExpression [Relation SimpleExpression].
SimpleExpression  =  ["+" | "-"] Term {AddOperator Term}.
Term              =  Factor {MulOperator Factor}.
Factor            =  Designator [ActualParameters]
                  |  number | character | string | NIL | Set
                  |  "(" Expression ")" | "~" Factor.
Set               =  "{" [Element {"," Element}] "}".
Element           =  Expression [".." Expression].
ActualParameters  =  "(" [ExpressionList] ")".
Relation          =  "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOperator       =  "+" | "-" | OR.
MulOperator       =  "*" | "/" | DIV | MOD | "&".
```

Einige Operatoren können auf Operanden mehrerer Typen angewendet werden und bedeuten dann verschiedene Operationen; die Bedeutung dieser Operatoren wird aufgrund der Typen ihrer Operanden bestimmt. Die Operanden müssen miteinander *ausdruckskompatibel* sein (siehe A.12.1).

Logische Operatoren

OR	logische Disjunktion	p OR q	≡	"if p then TRUE else q end"
&	logische Konjunktion	p & q	≡	"if p then q else FALSE end"
~	Negation	~ p	≡	"not p"

Logische Operatoren werden auf Operanden vom Typ BOOLEAN angewendet und liefern einen Wert vom Typ BOOLEAN.

Arithmetische Operatoren

+	Summe
-	Differenz
*	Produkt
/	reeller Quotient
DIV	ganzzahliger Quotient
MOD	Modulo-Operator

Die Operatoren +, -, * und / können auf *numerische* Operanden angewendet werden. Der Ergebnistyp ist bei +, - und * der Typ des Operanden, der den Typ des andern Operanden *einschließt*; bei / ist er der kleinste *reelle* Typ, der beide Operandentypen *einschließt*. Der monadische Operator - bedeutet Vorzeichenumkehrung; der monadische Operator + ist der Identitätsoperator. DIV und MOD können nur auf Operanden mit *ganzzahligem* Typ angewendet werden. Sie stehen über folgende Formeln miteinander in Beziehung (*y* ist positiv):

x = (x DIV y) * y + (x MOD y)
0 ≤ (x MOD y) < y

Beispiele:

x	y	x DIV y	x MOD y
5	3	1	2
-5	3	-2	1

Mengenoperatoren

+	Mengenvereinigung
-	Mengendifferenz (x - y = x * (-y))
*	Durchschnittsmenge
/	symmetrische Mengendifferenz (x / y = (x-y) + (y-x))

Mengenoperatoren können auf Operanden vom Typ SET angewendet werden und liefern ein Ergebnis vom Typ SET. Der monadische Operator - bezeichnet die Komplementärmenge, d.h. *-x* bezeichnet die Menge aller Zahlen zwischen MIN(SET) und MAX(SET), die nicht in *x* sind. Die Mengenoperatoren + und - sind *nicht* assoziativ *((a+b)-c ≠ a+(b-c))*.

Ein *Mengenkonstruktor* definiert den Wert einer Menge durch Aufzählung ihrer Elemente zwischen geschweiften Klammern. Die Elemente müssen von ganzzahligem Typ im Bereich MIN(SET) .. MAX(SET) sein. Der Bereich *a..b* enthält alle ganzen Zahlen, die größer oder gleich *a* und kleiner oder gleich *b* sind.

Vergleichsoperatoren

=	gleich
#	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
IN	Mengenzugehörigkeit
IS	Typtest

Vergleiche liefern ein Ergebnis vom Typ BOOLEAN. Die Operatoren =, #, <, <=, > und >= können auf *numerische* Typen, CHAR, (offene) 0X-terminierte Zeichenarrays und Zeichenketten angewendet werden, = und # zusätzlich auf BOOLEAN, SET, Zeiger- und Prozedurtypen (einschließlich des Werts NIL).

x IN *s* bedeutet "ist *x* ein Element von *s*?". *x* muß von einem *ganzzahligen* Typ sein und *s* vom Typ SET.

Der *Typtest v* IS *T* bedeutet "ist *v* vom dynamischen Typ *T* oder einer Erweiterung davon?". Er ist anwendbar, wenn

1. *v* ein Zeiger oder ein Var-Parameter mit Recordtyp ist und
2. *T* eine Erweiterung des statischen Typs von *v* ist.

Beispiele für Ausdrücke (siehe auch Beispiele in A.7):

```
1991                    INTEGER
i DIV 3                 INTEGER
~p OR q                 BOOLEAN
(i+j) * (i-j)           INTEGER
s - {8, 9, 13}          SET
i + x                   REAL
a[i+j] * a[i-j]         REAL
(0<=i) & (i<100)        BOOLEAN
t.key = 0               BOOLEAN
k IN {i..j-1}           BOOLEAN
w[i].name <= "John"     BOOLEAN
t IS CenterNode         BOOLEAN
```

A.9 Anweisungen

Anweisungen drücken Aktionen aus. Man unterscheidet zwischen einfachen und zusammengesetzten Anweisungen: *Einfache Anweisungen* sind die Zuweisung, der Prozeduraufruf, die Return-Anweisung und die Exit-Anweisung. *Zusammengesetzte Anweisungen* enthalten Teile, die selbst Anweisungen sind; sie dienen dazu, Anweisungsfolgen, Verzweigungen und Wiederholungen auszudrücken. Eine leere Anweisung bedeutet keine Aktion; sie erleichtert die Interpunktionsregeln in Anweisungsfolgen.

```
Statement =
    [ Assignment | ProcedureCall | IfStatement | CaseStatement
    | WhileStatement | RepeatStatement | ForStatement | LoopStatement
    | EXIT | RETURN [Expression] ].
```

A.9.1 Zuweisungen

Eine Zuweisung ersetzt den Wert einer Variablen durch den Wert eines Ausdrucks. Der Ausdruck muß mit der Variablen zuweisungskompatibel sein (siehe A.12.1). Der Zuweisungsoperator wird ":=" geschrieben und bedeutet "wird zu".

```
Assignment = Designator ":=" Expression.
```

Wird ein Ausdruck *e* mit Typ T_e einer Variablen *v* mit Typ T_v zugewiesen, geschieht das folgende:

1. Wenn T_v und T_e Recordtypen sind, werden nur diejenigen Felder von T_e zugewiesen, die auch zu T_v gehören (*Projektion*); der dynamische Typ von *v* muß *derselbe* sein wie der statische Typ von *v*; er wird durch die Zuweisung nicht verändert.
2. Wenn T_v und T_e Zeigertypen sind, ist der dynamische Typ von *v* nach der Zuweisung der dynamische Typ von *e*.
3. Wenn T_v ARRAY *n* OF CHAR ist und *e* eine Zeichenkette der Länge $m<n$, wird *v[i]* zu e_i für $i = 0..m-1$ und *v[m]* wird zu 0X.

Beispiele für Zuweisungen (siehe Beispiele in A.7):

```
i := 0
p := i = j
x := i + 1
k := Log2(i+j)              (siehe A.10.1)
f := Log2
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c
```

A.9.2 Prozeduraufrufe

Ein Prozeduraufruf aktiviert eine Prozedur. Er kann eine Liste *aktueller Parameter* aufweisen, die beim Aufruf die entsprechenden *formalen Parameter* der Prozedurdeklaration ersetzen (siehe A.10). Die Zuordnung der Parameter erfolgt aufgrund ihrer Position in der aktuellen und formalen Parameterliste. Man unterscheidet zwischen *Var-Parametern* und *Val-Parametern* (siehe A.10).

Wenn ein formaler Parameter ein Var-Parameter ist, muß der entsprechende aktuelle Parameter ein Bezeichner für eine Variable sein. Eventuelle Selektoren des Bezeichners werden vor der Ausführung der Prozedur ausgewertet.

Wenn ein formaler Parameter ein Val-Parameter ist, muß der entsprechende aktuelle Parameter ein Ausdruck sein. Er wird vor der Ausführung der Prozedur ausgewertet und dem formalen Parameter zugewiesen (siehe A.10.1).

```
ProcedureCall = Designator [ActualParameters].
```

Beispiele:

```
WriteInt(i*2+1)        (*siehe A.10.1*)
INC(w[k].count)
t.Insert("John")       (*siehe A.11*)
```

A.9.3 Anweisungsfolgen

Anweisungsfolgen stellen eine Folge von Aktionen dar. Sie bestehen aus den einzelnen Anweisungen, getrennt durch Strichpunkte.

```
StatementSequence = Statement {";" Statement}.
```

A.9.4 If-Anweisungen

```
IfStatement =
    IF Expression THEN StatementSequence
    { ELSIF Expression THEN StatementSequence }
    [ ELSE StatementSequence ]
    END.
```

If-Anweisungen drücken die bedingte Ausführung einer Anweisungfolge aus. Die booleschen Ausdrücke vor den Anweisungsfolgen werden in der Reihenfolge ihres Auftretens ausgewertet, bis einer von ihnen TRUE ergibt, worauf die zu ihm gehörende Anweisungsfolge ausgeführt wird. Wenn kein Ausdruck TRUE liefert, wird die Anweisungsfolge nach dem ELSE-Symbol ausgeführt, falls dieses existiert.

Beispiel:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = " ' ") OR (ch = ' " ') THEN ReadString
ELSE SpecialCharacter
END
```

A.9.5 Case-Anweisungen

Case-Anweisungen beschreiben die Auswahl einer Anweisungsfolge in Abhängigkeit vom Wert eines Ausdrucks. Nach Auswertung des *Case-Ausdrucks* wird diejenige Anweisungsfolge ausgeführt, deren *Case-Markenliste* den berechneten Wert enthält. Entspricht der Wert

des Ausdrucks keiner Case-Marke, wird die Anweisungsfolge nach dem ELSE-Symbol ausgeführt. Falls dieses fehlt, wird das Programm abgebrochen. Der Case-Ausdruck muß entweder von einem *ganzzahligen* Typ sein, der die Typen aller Case-Marken *einschließt* oder sowohl der Case-Ausdruck als auch die Case-Marken müssen vom Typ CHAR sein. Case-Marken sind Konstanten, deren Werte voneinander verschieden sein müssen.

```
CaseStatement  =  CASE Expression OF Case {"|" Case}
                  [ELSE StatementSequence] END.
Case           =  [CaseLabelList ":" StatementSequence].
CaseLabelList  =  CaseLabels {"," CaseLabels}.
CaseLabels     =  ConstExpression [".." ConstExpression].
```

Beispiel:

```
CASE ch OF
    "A".."Z": ReadIdentifier
|   "0".."9": ReadNumber
|   " ' ", ' " ': ReadString
ELSE SpecialCharacter
END
```

A.9.6 While-Anweisungen

While-Anweisungen beschreiben die wiederholte Ausführung einer Anweisungsfolge, solange der boolesche Ausdruck vor dieser Anweisungsfolge TRUE liefert. Der Ausdruck wird vor jeder Ausführung der Anweisungsfolge geprüft.

```
WhileStatement = WHILE Expression DO StatementSequence END.
```

Beispiel:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
```

A.9.7 Repeat-Anweisungen

Repeat-Anweisungen beschreiben die wiederholte Ausführung einer Anweisungsfolge. Sie wird mindestens einmal ausgeführt und so lange wiederholt, bis der boolesche Ausdruck am Ende der Repeat-Anweisung TRUE liefert.

```
RepeatStatement = REPEAT StatementSequence UNTIL Expression.
```

A.9.8 For-Anweisungen

For-Anweisungen beschreiben eine feste Anzahl von Ausführungen einer Anweisungsfolge, wobei eine *ganzzahlige* Variable (*Kontrollvariable*) bei jedem Durchlauf einen neuen Wert annimmt.

```
ForStatement =
    FOR ident ":=" Expression TO Expression [BY ConstExpression]
    DO StatementSequence END.
```

Die Anweisung

```
FOR v := low TO high BY step DO statements END
```

ist gleichbedeutend mit

```
temp := high; v := low;
IF step > 0 THEN
    WHILE v <= temp DO statements; v := v + step END
ELSE
    WHILE v >= temp DO statements; v := v + step END
END
```

temp hat *denselben* Typ wie *v* (siehe A.12.1). *step* muß ein *ganzzahliger* Konstantenausdruck sein, dessen Wert nicht 0 sein darf. Fehlt die Angabe der Schrittweite *step*, wird für sie der Wert 1 angenommen. Beispiele:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

A.9.9 Loop-Anweisungen

Loop-Anweisungen beschreiben die wiederholte Ausführung einer Anweisungsfolge, die durch die Ausführung einer Exit-Anweisung abgebrochen wird (siehe A.9.10).

```
LoopStatement = LOOP StatementSequence END
```

Beispiel:

```
LOOP
    ReadInt(i);
    IF i < 0 THEN EXIT END;
    WriteInt(i)
END
```

Loop-Anweisungen sind nützlich, um Schleifen mit mehreren Austrittspunkten darzustellen oder Schleifen, in denen sich die Austrittsbedingung in der Mitte der wiederholten Anweisungsfolge befindet.

A.9.10 Return- und Exit-Anweisungen

Eine Return-Anweisung bezeichnet den expliziten Aussprung aus einer Prozedur. Sie wird durch das Symbol RETURN ausgedrückt. Falls die Prozedur eine Funktionsprozedur ist, wird RETURN von einem Ausdruck gefolgt. Der Typ des Ausdrucks muß mit dem im Prozedurkopf angegebenen Ergebnistyp (siehe A.10) *zuweisungskompatibel* sein (siehe A.12.1).

Funktionsprozeduren *müssen* über eine Return-Anweisung verlassen werden, die das Ergebnis der Funktion zurückgibt. In gewöhnlichen Prozeduren wird nach der letzten Anweisung eine implizite Return-Anweisung eingefügt. Jede explizite Return-Anweisung ist daher ein zusätzlicher (meist durch eine Ausnahme bedingter) Aussprung.

Eine Exit-Anweisung wird durch das Symbol EXIT ausgedrückt. Sie bezeichnet den Aussprung aus der unmittelbar umgebenden Loop-Anweisung und die Fortsetzung mit der ersten Anweisung nach der Schleife. Exit-Anweisungen sind zwar nicht syntaktisch, aber durch ihren Kontext mit der Loop-Anweisung verbunden, die sie enthält.

A.9.11 With-Anweisungen

With-Anweisungen beschreiben die Ausführung einer Anweisungsfolge in Abhängigkeit vom Ergebnis eines *Typtests*. Innerhalb der Anweisungsfolge wird auf die getestete Variable eine *Typzusicherung* angewendet.

```
WithStatement  =  WITH Guard DO StatementSequence
                  { "|" Guard DO StatementSequence }
                  [ ELSE StatementSequence ] END.
Guard          =  Qualident ":" Qualident.
```

Wenn *v* eine Zeigervariable oder ein Var-Parameter mit Recordtyp ist und wenn ihr statischer Typ *T0* ist, bedeutet die Anweisung

```
WITH v: T1 DO S1
| v: T2 DO S2
ELSE S3
END
```

Falls *v* vom dynamischen Typ *T1* ist, wird *S1* ausgeführt, wobei *v* wie eine Variable mit statischem Typ *T1* behandelt wird; andernfalls, wenn *v* vom dynamischen Typ *T2* ist, wird *S2* ausgeführt, wobei *v* wie eine Variable mit statischem Typ *T2* behandelt wird; andernfalls wird *S3* ausgeführt. Wenn kein Typtest zutrifft und ein Else-Zweig fehlt, wird das Programm abgebrochen. *T1* und *T2* müssen Erweiterungen von *T0* sein. Beispiel:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

A.10 Prozedurdeklarationen

Eine Prozedurdeklaration besteht aus einem *Prozedurkopf* und einem *Prozedurrumpf*. Der Kopf definiert den Namen der Prozedur und ihre *formalen Parameter*, bei typgebundenen Prozeduren auch den *Empfängerparameter*. Der Rumpf enthält Deklarationen und Anweisungen. Der Prozedurname wird am Ende der Prozedurdeklaration wiederholt.

Es gibt zwei Arten von Prozeduren: *gewöhnliche Prozeduren* und *Funktionsprozeduren*. Funktionsprozeduren werden durch einen Bezeichner aktiviert, der Teil eines Ausdrucks ist; sie liefern ein Ergebnis, das als Operand in die Berechnung des Ausdrucks einfließt. Gewöhnliche Prozeduren werden durch einen Prozeduraufruf aktiviert. Funktionsprozeduren erkennt man daran, daß ihre formale Parameterliste einen Ergebnistyp enthält. Ihr Rumpf muß eine Return-Anweisung enthalten, die ihr Ergebnis bestimmt.

Alle in einem Prozedurrumpf deklarierten Konstanten, Variablen, Typen und Prozeduren sind lokal zu dieser Prozedur. Da Prozeduren selbst lokale Objekte sein können, dürfen Prozedurdeklarationen geschachtelt werden. Die Aktivierung einer Prozedur innerhalb ihres Rumpfes führt zu einem rekursiven Aufruf.

Die in umgebenden Blöcken deklarierten Objekte sind in jenen Teilen der Prozedur sichtbar in denen sie nicht durch lokal deklarierte Objekte gleichen Namens verdeckt werden.

```
ProcedureDeclaration   = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading       = PROCEDURE [Receiver] IdentDef
                         [FormalParameters].
ProcedureBody          = DeclarationSequence
                         [BEGIN StatementSequence] END.
DeclarationSequence    = {CONST {ConstDeclaration ";"}
                       | TYPE {TypeDeclaration ";"}
                       | VAR {VariableDeclaration ";"} }
                         {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
ForwardDeclaration     = PROCEDURE "^" [Receiver] IdentDef
                         [FormalParameters].
```

Wird in einer Prozedurdeklaration ein *Empfängerparameter* (receiver) deklariert, bedeutet das, daß die Prozedur an den Typ dieses Parameters gebunden ist (siehe A.10.2).

Eine Vorausdeklaration erlaubt, sich auf eine Prozedur zu beziehen, die erst später im Text deklariert wird. Die formalen Parameterlisten der Vorausdeklaration und der eigentlichen Deklaration müssen identisch sein.

A.10.1 Formale Parameter

Die in der formalen Parameterliste einer Prozedur deklarierten Namen heißen *formale Parameter*. Sie entsprechen den beim Prozeduraufruf angegebenen aktuellen Parametern. Die Zuordnung zwischen formalen und aktuellen Paremetern findet beim Prozeduraufruf statt. *Var-Parameter* werden mit dem Schlüsselwort VAR deklariert, *Val-Parameter* ohne dieses Schlüsselwort. Val-Parameter sind lokale Variablen, denen der Wert des entsprechenden aktuellen Parameters als Anfangswert zugewiesen wird. Var-Parameter entsprechen aktuellen Parametern, die Variablen sind; sie stehen für diese Variablen. Der Sichtbarkeitsbereich eines formalen Parameters erstreckt sich von seiner Deklaration bis zum Ende des Prozedurblocks, in dem er deklariert ist. Eine parameterlose Funktionsprozedur muß eine leere Parameterliste aufweisen; bei ihrem Aufruf muß ebenfalls eine leere Parameterliste angegeben werden. Der Ergebnistyp einer Funktionsprozedur kann weder ein Record noch ein Array sein.

```
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection = [VAR] ident {"," ident} ":" Type.
```

Wenn T_f der Typ eines formalen Parameters f ist (kein offenes Array) und T_a der Typ des entsprechenden aktuellen Parameters a, so gilt: Bei Var-Parametern muß T_f und T_a *derselbe* Typ sein oder T_f muß ein Recordtyp sein und T_a eine Erweiterung davon. Bei Val-Parametern muß a *zuweisungskompatibel* mit f sein (siehe A.12.1).

Wenn T_f ein offenes Array ist, muß a *arraykompatibel* mit f sein (siehe A.12.1). Die Längen von f werden aus a übernommen.

Beispiele für Prozedurdeklarationen:

```
PROCEDURE ReadInt (VAR x: INTEGER);
   VAR i: INTEGER; ch: CHAR;
BEGIN
   i := 0; Read(ch);
   WHILE ("0" <= ch) & (ch <= "9") DO
      i := 10*i + (ORD(ch) - ORD("0")); Read(ch)
   END;
   x := i
END ReadInt;

PROCEDURE WriteInt (x: INTEGER); (*0 <= x < 10000*)
   VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN
   i := 0;
   REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i) UNTIL x = 0;
   REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt;

PROCEDURE WriteString (s: ARRAY OF CHAR);
   VAR i: INTEGER;
BEGIN
   i := 0; WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString;

PROCEDURE Log2 (x: INTEGER): INTEGER; (*x>0*)
   VAR y: INTEGER;
BEGIN
   y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
   RETURN y
END Log2;
```

A.10.2 Typgebundene Prozeduren

Prozeduren können einem im selben Block deklarierten Recordtyp zugeordnet werden (*typgebundene Prozeduren*). Die Zuordnung erfolgt aufgrund des Typs des *Empfängerparameters* im Prozedurkopf. Der Empfänger kann entweder ein Var-Parameter mit Recordtyp *T* oder ein Val-Parameter mit Zeigertyp POINTER TO *T* sein. Die Prozedur gehört zum Typ *T* und ist lokal zu diesem.

```
ProcedureHeading  =  PROCEDURE [Receiver] IdentDef
                     [FormalParameters].
Receiver          =  "(" [VAR] ident ":" ident ")".
```

Wenn eine Prozedur *P* zu einem Typ *T0* gehört, gehört sie auch zu jedem Typ *T1*, der eine Erweiterung von *T0* ist. Eine gleichnamige Prozedur *P'* darf jedoch ausdrücklich *T1* zugeordnet werden; sie *über-*

schreibt die an *T0* gebundene Prozedur, d.h. sie ersetzt *P* für *T1*. Die formalen Parameter von *P* und *P'* müssen *übereinstimmen* (A.12.1). Wird *P* und *T1* exportiert (A.4), muß auch *P'* exportiert werden.

Wenn *v* ein Bezeichner ist und *P* eine typgebundene Prozedur, bezeichnet *v.P* diejenige Prozedur, die zum dynamischen Typ von *v* gehört (*dynamische Bindung*); das kann eine andere Prozedur sein als die zum statischen Typ von *v* gehörige. *v* wird gemäß den in Kapitel A.10.1 beschriebenen Parameterübergaberegeln an den Empfängerparameter von *P* übergeben.

Wenn *v* ein Empfängerparameter vom Typ *T* (oder POINTER TO *T*) ist, bezeichnet *v.P^* die überschriebene Prozedur *P*, die zum Basistyp von *T* gehört.

In einer Vorausdeklaration einer typgebundenen Prozedur muß der Empfängerparameter *denselben* Typ haben wie in der eigentlichen Deklaration. Die formalen Parameterlisten beider Deklarationen müssen identisch sein. Beispiele:

```
PROCEDURE (t: Tree) Insert (node: Tree);
   VAR p, father: Tree;
BEGIN
   p := t;
   REPEAT father := p;
      IF node.key = p.key THEN RETURN END;
      IF node.key < p.key THEN p := p.left ELSE p := p.right END
   UNTIL p = NIL;
   IF node.key < father.key THEN father.left := node
   ELSE father.right := node
   END;
   node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree); (*redefinition*)
BEGIN
   WriteInt(node(CenterNode).width);
   t.Insert^ (node)
END Insert;
```

A.10.3 Vordeklarierte Prozeduren

Die folgenden Tabellen beschreiben vordeklarierte Prozeduren. Einige sind generisch, d.h. auf verschiedene Operandentypen anwendbar. *v* steht für eine Variable, *x* und *n* für Ausdrücke und *T* für einen Typ.

Name	*Operandentyp*	*Ergebnistyp*	*Bedeutung*
ABS(x)	numerisch	Typ von x	Absolutwert
ASH(x, n)	x, n: ganzzahlig	LONGINT	arithmetische Shift-Operation ($x * 2^n$)
CAP(x)	CHAR (Buchstabe)	CHAR	entsprechender Großbuchstabe
CHR(x)	ganzzahlig	CHAR	Zeichen mit Ordinalwert x
ENTIER(x)	reell	LONGINT	größte ganze Zahl kleiner oder gleich x
LEN(v, n)	v: Array n: ganzz. Konstante	LONGINT	Länge von v in der Dimension n (erste Dimension = 0)
LEN(v)	v: Array	LONGINT	äquivalent zu LEN(v, 0)
LONG(x)	SHORTINT INTEGER REAL	INTEGER LONGINT LONGREAL	Identität
MAX(T)	T: Standardtyp T: SET	T INTEGER	größter Wert des Typs T größtes Element des Typs SET
MIN(T)	T: Standardtyp T: SET	T INTEGER	kleinster Wert des Typs T 0
ODD(x)	ganzzahlig	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	Ordinalwert von x
SHORT(x)	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	Identität Identität Identität (mögl. Genauigkeitsverlust)
SIZE(T)	beliebiger Typ	ganzzahlig	Länge von T in Bytes

Name	*Operandentyp*	*Bedeutung*
ASSERT(x [, n])	x: boolescher Ausdruck	Programmabbruch falls ~x
COPY(x, v)	x: Zeichenarray oder Zeichenkette v: Zeichenarray	v := x
DEC(v)	ganzzahlig	v := v - 1
DEC(v, n)	v, n: ganzzahlig	v := v - n
EXCL(v, n)	v: SET; x: ganzzahlig	v := v - {x}
HALT(x)	ganzzahlige Konstante	Programmabbruch
INC(v)	ganzzahlig	v := v + 1
INC(v, n)	v, n: ganzzahlig	v := v + n
INCL(v, n)	v: SET; x: ganzzahlig	v := v + {x}
NEW(v)	Zeiger auf Rec. oder Array fester Länge	Anlegen von v^
NEW(v, x_0, ...x_n)	v: Zeiger auf offenes Array; x_i ganzzahlig	Anlegen von v^mit Längen $x_0..x_n$

COPY weist eine Zeichenkette oder ein 0X-terminiertes Zeichenarray einem anderen Zeichenarray zu. Der zugewiesene Wert wird nötigenfalls auf die Länge des Zieloperanden minus 1 gekürzt. Der Zieloperand wird immer durch das Zeichen 0X abgeschlossen. Bei ASSERT(*x*, *n*) und HALT(*n*) wird die Interpretation von *n* dem zugrundeliegenden Betriebssystem überlassen (z.B. Fehlernummer).

A.11 Module

Ein Modul ist eine Sammlung von Deklarationen für Konstanten, Typen, Variablen und Prozeduren, zusammen mit einer Anweisungsfolge, die vor allem dazu dient, den Variablen Anfangswerte zu geben. Ein Modul kann für sich übersetzt werden.

```
Module     =  MODULE ident ";" [ImportList] DeclarationSequence
              [BEGIN StatementSequence] END ident ".".
ImportList =  IMPORT Import {"," Import} ";".
Import     =  [ident ":="] ident.
```

Die *Importliste* enthält die Namen der importierten Module. Wenn ein Modul *M* ein Modul *A* importiert und *A* einen Namen *x* exportiert, wird *x* in *M* als *A.x* angesprochen. Wird *A* in der Form *B := A* importiert, spricht man *x* als *B.x* an; das erlaubt die Verwendung kurzer qualifizierter Namen. Namen, die exportiert und damit in anderen Modulen sichtbar sein sollen, müssen bei ihrer Deklaration mit einer Exportmarke versehen werden (siehe A.4).

Die Anweisungsfolge nach dem Symbol BEGIN (*Modulrumpf*) wird ausgeführt, wenn das Modul geladen wird. Vorher werden alle importierten Module geladen. Daraus folgt, daß zyklischer Import von Modulen verboten ist. Parameterlose, exportierte Prozeduren heißen *Kommandos* und können als Programme aufgerufen werden (siehe A.12.4).

```
MODULE Trees;
   IMPORT Texts, Oberon;
   (* exportiert: Tree, Node, Insert, Search, Write, NewTree *)
   (* exportiert schreibgeschützt: Node.name *)

   TYPE
      Tree* = POINTER TO Node;
      Node* = RECORD
         name-: POINTER TO ARRAY OF CHAR;
         left, right: Tree
      END;

   VAR w: Texts.Writer;

   PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
      VAR p, father: Tree;
   BEGIN
      p := t;
      REPEAT father := p;
         IF name = p.name^ THEN RETURN END;
         IF name < p.name^ THEN p := p.left ELSE p := p.right END
      UNTIL p = NIL;
```

```
    NEW(p); p.left := NIL; p.right := NIL;
    NEW(p.name, LEN(name)+1); COPY(name, p.name^);
    IF name < father.name^ THEN father.left := p ELSE father.right := p END
  END Insert;

  PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
    VAR p: Tree;
  BEGIN
    p := t;
    WHILE (p # NIL) & (name # p.name^) DO
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    END;
    RETURN p
  END Search;

  PROCEDURE (t: Tree) Write*;
  BEGIN
    IF t.left # NIL THEN t.left.Write END;
    Texts.WriteString(w, t.name^); Texts.WriteLn(w);
    Texts.Append(Oberon.Log, w.buf);
    IF t.right # NIL THEN t.right.Write END
  END Write;

  PROCEDURE NewTree* (name: ARRAY OF CHAR): Tree;
    VAR t: Tree;
  BEGIN
    NEW(t); NEW(t.name, LEN(name)+1); COPY(name, t.name^);
    t.left := NIL; t.right := NIL; RETURN t
  END NewTree;

BEGIN
  Texts.OpenWriter(w)
END Trees.
```

A.12 Anhänge zur Sprachdefinition

A.12.1 Begriffsdefinitionen

Ganzzahlige Typen	SHORTINT, INTEGER, LONGINT
Reelle Typen	REAL, LONGREAL
Numerische Typen	*Ganzzahlige* Typen und *reelle* Typen

Derselbe Typ

Zwei Variablen *a* und *b* mit den Typen T_a und T_b haben *denselben* Typ, wenn eine der folgenden Aussagen gilt:

1. T_a und T_b werden durch denselben Typnamen bezeichnet.
2. T_a ist deklariert als $T_a = T_b$.
3. *a* und *b* treten in der gleichen Namenliste einer Variablen-, Parameter- oder Recordfeld-Deklaration auf und sind keine offenen Arrays.

Der gleiche Typ

Zwei Typen T_a und T_b sind *gleich*, wenn eine der folgenden Aussagen gilt:

1. T_a und T_b sind derselbe Typ.
2. T_a und T_b sind offene Arraytypen, deren Elementtypen *gleich* sind.
3. T_a und T_b sind Prozedurtypen, deren formale Parameterlisten *übereinstimmen*.

Typeinschluß

Numerische Typen *schließen* sich und die Werte kleinerer *numerischer* Typen gemäß folgender Hierarchie *ein*:

LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

Typerweiterung und Basistyp

In einer Typdeklaration T_b = RECORD (T_a) ... END ist T_b eine direkte Erweiterung von T_a und T_a ist der direkte Basistyp von T_b. Allgemein ist ein Typ T_b eine Erweiterung eines Typs T_a (T_a ist ein Basistyp von T_b) wenn eine der folgenden Aussagen gilt:

1. T_a und T_b sind *dieselben* Typen.
2. T_b ist eine direkte Erweiterung einer Erweiterung von T_a.

Wenn P_a = POINTER TO T_a und P_b = POINTER TO T_b gilt, wobei T_b eine Erweiterung von T_a ist, so ist auch P_b eine Erweiterung von P_a und P_a ist der Basistyp von P_b.

Zuweisungskompatibilität

Ein Ausdruck *e* vom Typ T_e ist *zuweisungskompatibel* mit einer Variablen *v* vom Typ T_v wenn eine der folgenden Bedingungen gilt:

1. T_e und T_v sind *dieselben* Typen.
2. T_e und T_v sind *numerische* Typen und T_v *schließt* T_e *ein.*
3. T_e und T_v sind Recordtypen, T_e ist eine Erweiterung von T_v und der dynamische Typ von v ist T_v.
4. T_e und T_v sind Zeigertypen und T_e ist eine Erweiterung von T_v.
5. T_v ist ein Zeigertyp oder ein Prozedurtyp und e ist NIL.
6. T_v ist ARRAY n OF CHAR, e ist eine Zeichenkettenkonstante der Länge m und $m < n$.
7. T_v ist ein Prozedurtyp und e ist der Name einer Prozedur, deren formale Parameter mit denen von T_v *übereinstimmen.*

Array-kompatibilität

Ein aktueller Parameter a vom Typ T_a ist mit einem formalen Parameter f vom Typ T_f *arraykompatibel*, wenn eine der Bedingungen gilt:

1. T_f und T_a sind *dieselben* Typen.
2. T_f ist ein offenes Array, T_a ist ein beliebiges Array und ihre Elementtypen sind *arraykompatibel.*
3. f ist ein Val-Parameter vom Typ ARRAY OF CHAR und a ist eine Zeichenkettenkonstante.

Ausdrucks-kompatibilität

Die Operandentypen eines gegebenen Operators sind *ausdruckskompatibel*, wenn sie folgender Tabelle entsprechen (die auch den Ergebnistyp des Ausdrucks zeigt). *T1* muß eine Erweiterung von *T0* sein. Zeichenarrays in einem Vergleichsausdruck müssen 0X terminiert sein.

Operator	*Erster Operand*	*Zweiter Operand*	*Ergebnistyp*
+ - *	numerisch	numerisch	kleinster numerischer Typ, der beide Operandentypen einschließt
/	numerisch	numerisch	kleinster reller Typ, der beide Operandentypen einschließt
+ - * /	SET	SET	SET
DIV MOD	ganzzahlig	ganzzahlig	kleinster ganzzahliger Typ, der beide Operandentypen einschließt
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	numerisch	numerisch	BOOLEAN
	CHAR	CHAR	BOOLEAN
	Zeichenarry, -kette	Zeichenarray, -kette	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, Zeiger T0 oder T1	NIL, Zeiger T0 oder T1	BOOLEAN
	NIL, Prozedurtyp	NIL, Prozedurtyp	BOOLEAN
IN	ganzzahlig	SET	BOOLEAN
IS	T0	T1	BOOLEAN

Überein-stimmende Parameterlisten

Zwei formale Parameterlisten *stimmen überein*, falls sämtliche der folgenden Bedingungen zutreffen:

1. Sie enthalten dieselbe Anzahl Parameter.
2. Sie haben *denselben* Funktions-Ergebnistyp oder keinen.
3. Parameter an gleichen Positionen haben *gleiche* Typen.
4. Parameter an gleichen Pos. sind beide Val- oder Var-Parameter.

A.12.2 Syntax von Oberon-2

```
Module      =  MODULE ident ";" [ImportList] DeclSeq  [BEGIN StatSeq] END ident ".".
ImportList  =  IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".
DeclSeq     =  { CONST {ConstDecl ";" } | TYPE {TypeDecl ";"} | VAR {VarDecl ";"}}
               {ProcDecl ";" | ForwardDecl ";"}.
ConstDecl   =  IdentDef "=" ConstExpr.
TypeDecl    =  IdentDef "=" Type.
VarDecl     =  IdentList ":" Type.
ProcDecl    =  PROCEDURE [Receiver] IdentDef [FormalPars] ";" DeclSeq [BEGIN StatSeq] END ident.
ForwardDecl =  PROCEDURE "^" [Receiver] IdentDef [FormalPars].
FormalPars  =  "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection   =  [VAR] ident {"," ident} ":" Type.
Receiver    =  "(" [VAR] ident ":" ident ")".
Type        =     Qualident
            |  ARRAY [ConstExpr {"," ConstExpr}] OF Type
            |  RECORD ["("Qualident")"] FieldList {";" FieldList} END
            |  POINTER TO Type
            |  PROCEDURE [FormalPars].
FieldList   =  [IdentList ":" Type].
StatSeq     =  Stat {";" Stat}.
Stat    =  [   Designator ":=" Expr
           |   Designator ["(" [ExprList] ")"]
           |   IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq} [ELSE StatSeq] END
           |   CASE Expr OF Case {"|" Case} [ELSE StatSeq] END
           |   WHILE Expr DO StatSeq END
           |   REPEAT StatSeq UNTIL Expr
           |   FOR ident ":=" Expr TO Expr [BY ConstExpr] DO StatSeq END
           |   LOOP StatSeq END
           |   WITH Guard DO StatSeq {"|" Guard DO StatSeq} [ELSE StatSeq] END
           |   EXIT
           |   RETURN [Expr]
           ].
Case        =  [CaseLabels {"," CaseLabels} ":" StatSeq].
CaseLabels  =  ConstExpr [".." ConstExpr].
Guard       =  Qualident ":" Qualident.
ConstExpr   =  Expr.
Expr        =  SimpleExpr [Relation SimpleExpr].
SimpleExpr  =  ["+" | "-"] Term {AddOp Term}.
Term        =  Factor {MulOp Factor}.
Factor      =  Designator ["(" [ExprList] ")"] | number | character | string | NIL | Set | "(" Expr ")" | " ~ " Factor.
Set         =  "{" [Element {"," Element}] "}".
Element     =  Expr [".." Expr].
Relation    =  "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp       =  "+" | "-" | OR.
MulOp       =  " * " | "/" | DIV | MOD | "&".
Designator  =  Qualident {"." ident | "[" ExprList "]" | " ^ " | "(" Qualident ")"}.
ExprList    =  Expr {"," Expr}.
IdentList   =  IdentDef {"," IdentDef}.
Qualident   =  [ident "."] ident.
IdentDef    =  ident [" * " | "-"].
```

A.12.3 Modul SYSTEM

Das Pseudo-Modul SYSTEM enthält Typen und Prozeduren, mit denen sich systemnahe Operationen für einen bestimmten Rechner oder ein bestimmtes Betriebssystem implementieren lassen. Darunter fallen Operationen zum Ansprechen peripherer Geräte oder zur Umgehung der Typregeln der Sprache. Es wird dringend empfohlen, SYSTEM nur in wenigen, systemnahen Modulen zu verwenden. Solche Module sind von Natur aus unportabel. Man erkennt sie daran, daß sie SYSTEM importieren. Die folgenden Angaben gelten für die Implementierung von Oberon-2 unter Windows.

SYSTEM exportiert einen Typ BYTE mit folgenden Eigenschaften: Variablen vom Typ CHAR oder SHORTINT können Variablen vom Typ BYTE zugewiesen werden. Wenn ein formaler Var-Parameter vom Typ ARRAY OF BYTE ist, darf der aktuelle Parameter ein beliebiger Typ sein.

Ein anderer von SYSTEM exportierter Typ ist PTR. Einer Variablen vom Typ PTR kann eine Variable eines beliebigen Zeigertyps zugewiesen werden. Wenn ein formaler VAR-Parameter vom Typ PTR ist, darf der aktuelle Parameter ein beliebiger Zeiger sein.

Die von SYSTEM exportierten Prozeduren sind in folgenden Tabellen beschrieben. Die meisten werden in einen einzigen Maschinenbefehl übersetzt, der an der Stelle ihres Aufrufs in das Programm eingefügt wird. v steht für eine Variable, *x*, *y*, *a* und *n* für Ausdrücke und *T* für einen Typ.

Name	*Operandentypen*	*Ergebnistyp*	*Bedeutung*
ADR(v)	beliebig	LONGINT	Adresse der Variablen v
BIT(a, n)	a: LONGINT; n: ganzzahlig	BOOLEAN	Bit n von Mem[a]
CC(n)	ganzzahlige Konstante	BOOLEAN	Bedingungsbit n ($0 \leq n \leq 15$)
LSH(x, n)	x: ganzzahlig, CHAR, BYTE; n: ganzzahlig	Typ von x	logisches Shift
ROT(x, n)	x: ganzzahlig, CHAR, BYTE; n: ganzzahlig	Typ von x	Rotation
VAL(T, x)	T, x: beliebig	T	Typ von x wird in T umgewandelt

Name	*Operandentypen*	*Bedeutung*
GET(a, v)	a: LONGINT; v: Standardtyp, Zeiger-, Prozedurtyp	v := Mem[a]
PUT(a, x)	a: LONGINT; x: Standardtyp, Zeiger-, Prozedurtyp	Mem[a] := x
GETREG(n, v)	n: ganzzahlige Konstante; v: Standardtyp, Zeiger-, Prozedurtyp	v := Register n
PUTREG(n, x)	n: ganzzahlige Konstante; v: Standardtyp, Zeiger-, Prozedurtyp	Register n := x
MOVE(a0, a1, n)	a0, a1: LONGINT; n: ganzzahlig	Mem[a1..a1+n-1] := Mem[a0..a0+n-1]
NEW(v, n)	v: beliebiger Zeiger; n: ganzzahlig	legt einen Block mit n Bytes an und weist seine Adresse v zu

A.12.4 Die Oberon-Umgebung

Oberon-2-Programme laufen üblicherweise in einer Umgebung (hier Oberon-System genannt), die Kommandoaktivierung, Speicherbereinigung (garbage collection), dynamisches Laden von Modulen und gewisse Laufzeitdatenstrukturen zur Verfügung stellt. Obwohl diese Umgebung nicht Teil der Sprache ist, trägt sie wesentlich zur Mächtigkeit von Oberon-2 bei und wird zu einem gewissen Grad von der Sprache vorausgesetzt. Dieses Kapitel beschreibt die wesentlichen Eigenschaften einer typischen Oberon-Umgebung und gibt Hinweise für ihre Implementierung. Details können in [WiG92], [Rei91] und [PHT91] gefunden werden.

Kommandos

Ein Kommando ist eine parameterlose Prozedur *P*, die von einem Modul *M* exportiert wird. Es wird mit *M.P* angesprochen und kann unter diesem Namen vom Benutzer im Dialog mit dem Betriebssystem aufgerufen werden (z.B. durch Anklicken des Namens). In Oberon ruft man Kommandos anstatt Programme auf. Das gibt dem Benutzer eine feinere Auswahlmöglichkeit für die Aktivierung von Abläufen und erlaubt Programme mit mehreren Eintrittspunkten.

Beim Aufruf eines Kommandos *M.P* wird das Modul *M* geladen (falls es nicht bereits geladen ist) und die Prozedur *P* ausgeführt. Ist *P* beendet, wird die Kontrolle an das Oberon-System zurückgegeben, *M* bleibt aber geladen. Alle globalen Variablen und alle dynamischen Datenstrukturen, die über globale Zeiger in *M* erreicht werden können, behalten ihre Werte. Wird P *oder* ein anderes Kommando aus *M* erneut aufgerufen, kann es diese Werte weiterbenutzen.

Das folgende Modul zeigt, wie Kommandos benutzt werden. Es implementiert eine abstrakte Datenstruktur *Counter*, die einen Zähler

kapselt und Kommandos zum Erhöhen und Drucken des Zählers zur Verfügung stellt.

```
MODULE Counter;
IMPORT In, Out;

VAR
   counter: LONGINT;

PROCEDURE Add*; (*takes a numeric argument from the command line*)
   VAR n: INTEGER;
BEGIN
   In.Open; In.Int(n);
   IF In.Done THEN INC(counter, n) END
END Add;

PROCEDURE Write*;
BEGIN
   Out.Int(counter, 5); Out.Ln
END Write;

BEGIN
   counter := 0
END Counter.
```

Der Benutzer kann die folgenden beiden Kommandos aufrufen:

Counter.Add n	addiert n zum Zähler
Counter.Write	gibt den Zähler am Bildschirm aus

Da Kommandos parameterlos sind, müssen sie sich ihre Parameter auf andere Art besorgen. Im allgemeinen können Kommandos ihre Parameter von einer beliebigen Stelle holen (zum Beispiel vom Text, der dem Kommando folgt, von der jüngsten Selektion oder vom Inhalt eines markierten Fensters). Das Kommando *Add* benutzt das Modul *In*, um die Zahl *n* im Text hinter dem Kommando zu lesen.

Jeder Aufruf von *Counter.Add* erhöht den Wert der Variablen *counter* um *n*. Jeder Aufruf von *Counter.Write* gibt den momentanen Wert von *counter* am Bildschirm aus. Vor der ersten Ausführung eines Kommandos aus *Counter* wird das Modul geladen und sein Rumpf ausgeführt.

Das Modul *Counter* bleibt nun geladen, bis es mit dem Oberon-Kommando *System.Free Counter ~* ausdrücklich aus dem Hauptspeicher entfernt wird (zum Beispiel um die geladene Version durch eine neue zu ersetzen).

Dynamisches Laden von Modulen

Das Oberon-System erlaubt es, ein Kommando eines noch ungeladenen Moduls zu aktivieren. Dieses Modul wird dann dynamisch geladen und zu anderen, bereits geladenen Modulen gebunden; anschließend wird das gewünschte Kommando ausgeführt.

Dynamisches Laden ermöglicht es, ein Programm in einer Grundversion zu starten und es zur Laufzeit durch Hinzuladen neuer Module zu erweitern, falls sich die Notwendigkeit dafür ergibt.

Ein Modul *M0* kann das dynamische Laden eines Moduls *M1* bewirken, das es zur Übersetzungszeit nicht kennt, sondern dessen Namen es zur Laufzeit als Parameter bekommt. *M1* kann *M0* importieren und benutzen, umgekehrt muß *M0* aber statisch nicht wissen, daß *M1* existiert. *M1* kann ein Modul sein, das lange nach *M0* entworfen und implementiert wurde (Abb. A.1).

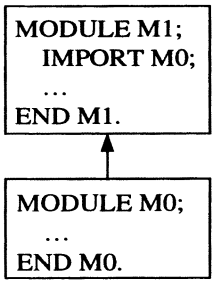


Abb. A.1 *M0* kann *M1* dynamisch laden
(der Pfeil bedeutet eine Import-Beziehung)

Speicherbereinigung

In Oberon-2 kann man mit der vordeklarierten Prozedur NEW Variablen im Freispeicherbereich anlegen. Der Programmierer gibt den Speicherplatz einer Variablen aber nie explizit frei. Stattdessen benutzt das Oberon-System eine *automatische Speicherbereinigung* (*garbage collection*), die alle Speicherbereiche, die nicht mehr über Zeiger erreichbar sind, wieder verfügbar macht.

Die Speicherbereinigung befreit den Programmierer von der fehleranfälligen Aufgabe, nicht mehr benötigte Datenstrukturen korrekt freizugeben. Dies ist besonders in objektorientierten Programmen wichtig, in denen der Programmierer auf Grund der Datenkapselung nie weiß, ob nicht in irgendeinem Objekt noch versteckte Zeiger auf ein freizugebendes Objekt existieren.

Browser

Die Schnittstelle eines Moduls (die Deklarationen der exportierten Namen) wird durch einen sogenannten *Browser* aus dem Quelltext des Moduls gewonnen. Aus dem Modul *Trees* (siehe A.11) erzeugt das Kommando *Browser.ShowDef Trees* folgende Schnittstelle.

```
DEFINITION Trees;
TYPE
   Tree = POINTER TO Node;
   Node = RECORD
      name-: POINTER TO ARRAY OF CHAR;
      PROCEDURE (t: Tree) Insert (name: ARRAY OF CHAR);
      PROCEDURE (t: Tree) Search (name: ARRAY OF CHAR): Tree;
      PROCEDURE (t: Tree) Write;
   END;
PROCEDURE NewTree (): Tree;
END Trees.
```

Bei einem Recordtyp sammelt der Browser auch alle zu diesem Typ gehörenden Prozeduren und zeigt sie in der Recorddeklaration an.

Typinformationen zur Laufzeit

Zur Laufzeit müssen gewisse Informationen über Records vorhanden sein: Für Typtests und Typzusicherungen wird der dynamische Typ von Recordvariablen benötigt; für den Aufruf typgebundener Prozeduren braucht man eine Tabelle mit den Adressen dieser Prozeduren. Schließlich braucht man zur Speicherbereinigung Informationen über die Position von Zeigern in Records. Alle diese Informationen werden in sogenannten *Typdeskriptoren* gespeichert, von denen es einen pro Recordtyp gibt.

Der dynamische Typ eines Records entspricht der Adresse seines Typdeskriptors. Bei dynamisch erzeugten Records wird diese Adresse in einer sogenannten *Typmarke* gespeichert, die vor den eigentlichen Recordfeldern liegt und für den Programmierer unsichtbar ist. Wenn *t* eine Variable vom Typ *CenterTree* ist (siehe Beispiele in A.6) zeigt Abb. A.2 eine mögliche Implementierung der Laufzeit-Typinformationen.

Sowohl die Prozedurentabelle als auch die Tabelle der Zeigerpositionen können bei einer Erweiterung des Typs wachsen. Daher sind sie an den entgegengesetzten Enden des Typdeskriptors angebracht und wachsen in entgegengesetzte Richtungen.

Eine typgebundene Prozedur *t.P* wird mit der Adresse *t.Typmarke.Prozedurentabelle[Index$_p$]* aufgerufen. Der Prozedurtabel-

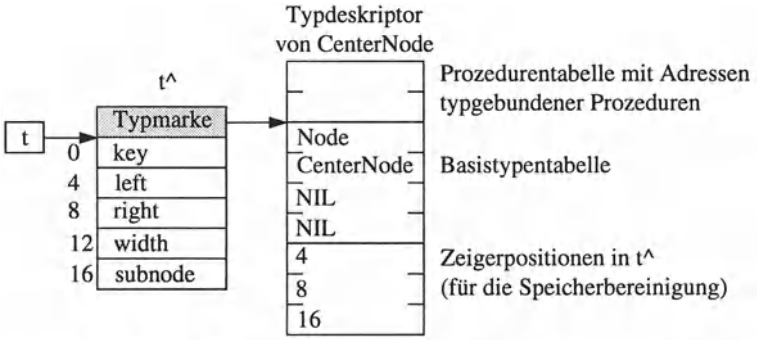


Abb. A.2 Eine Variable *t* vom Typ *CenterTree*, das Record *t^* vom Typ *CenterNode*, auf das sie zeigt, und dessen Typdeskriptor

lenindex jeder typgebundenen Prozedur ist zur Übersetzungszeit bekannt.

Ein Typtest *v IS T* wird übersetzt *als v.Typmarke.Basistypen [Erweiterungsstufe$_T$] = Typdeskriptoradresse$_T$*. Sowohl die Erweiterungsstufe eines Typs *T* als auch die Adresse seines Typdeskriptors sind zur Übersetzungszeit bekannt. Die Erweiterungsstufe von *Node* ist zum Beispiel 0 (*Node* hat keinen Basistyp), die von *CenterNode* ist 1.

B Bibliotheksmodule

Dieser Anhang beschreibt die in diesem Buch verwendeten Bibliotheksmodule des Oberon-Systems.

Modul	*Aufgabe*
In	Formatierte Eingabe von Zahlen, Namen, Zeichen und Zeichenketten.
Out	Formatierte Ausgabe von Zahlen, Zeichen und Zeichenketten.
Modules	Verwaltung der geladenen Module.
Types	Verwaltung von Record-Typen zur Laufzeit.
OS	*OS* ist kein Modul des Oberon-Systems, sondern ein Deckelmodul für verschiedene andere Module (*Display*, *Files*, *Input*, *Fonts*, *Oberon*), auf die in diesem Buch aus Platzgründen nicht näher eingegangen wird. Interessierte Leser seien auf [Rei91] verwiesen. Die Implementierung von *OS* ist Teil der Quellprogramme der Fallstudie aus Kapitel 12 (siehe Begleit-CD).

Weitere Informationen über Oberon-Bibliotheksmodule findet man in [Rei91], [ReW92] und [WiG92]. Die in Anhang D beschriebenen Oberon-Implementierungen enthalten ebenfalls On-Line-Dokumentation zu den Bibliotheksmodulen.

B.1
Modul In

Modul *In* bietet Prozeduren zur formatierten Eingabe von Zeichen, Zeichenfolgen, Zahlen und Namen. Es liest von einem Eingabestrom, dessen Position vom Benutzer gesetzt werden kann. Die hier beschriebene Version von *In* ist eine Erweiterung der Version aus [Rei91].

```
DEFINITION In;
   IMPORT Texts;

   CONST
      (*symbol codes*)
      inval = 0; name = 1; string = 2; int = 3; real = 4; longReal = 5; char = 6;

   VAR
      Done: BOOLEAN;

   PROCEDURE Open;
   PROCEDURE OpenText (t: Texts.Text; pos: LONGINT);

   PROCEDURE Next(): INTEGER;

   PROCEDURE Char (VAR ch: CHAR);
   PROCEDURE Int (VAR i: INTEGER);
   PROCEDURE LongInt (VAR i: LONGINT);
   PROCEDURE Real (VAR x: REAL);
   PROCEDURE LongReal (VAR y: LONGREAL);
   PROCEDURE String (VAR str: ARRAY OF CHAR);
   PROCEDURE Name (VAR name: ARRAY OF CHAR);
END In.
```

Zustand

- *Leseposition*. Die Position im Eingabestrom, von der das nächste Symbol gelesen wird. *Open* setzt sie an den Anfang des Eingabestroms, *OpenText* setzt sie an eine bestimmte Stelle eines Textes. Nachdem ein Symbol gelesen wurde, befindet sich die Leseposition unmittelbar nach diesem Symbol. Vor dem ersten Aufruf von *Open* oder *OpenText* ist die Leseposition undefiniert.

- *Done*. Wenn *Done* nach einer Leseoperation TRUE ist, war die Operation erfolgreich und das Resultat ist gültig. Eine erfolglose Leseoperation setzt *Done* auf FALSE (z.B. wenn versucht wird, über das Ende des Eingabestroms hinaus zu lesen). Ist *Done* einmal FALSE, wird dieser Wert bis zum nächsten Aufruf von *Open* oder *OpenText* beibehalten.

Operationen

- *Open* setzt die Leseposition an den Anfang des Eingabestroms. Der Eingabestrom ist der Text unmittelbar hinter dem zuletzt aufgerufenen Kommando. Wenn dieser Text mit dem Zeichen ^ beginnt, wird die Leseposition an den Anfang der jüngsten Selektion gesetzt (falls keine existiert, ist *Done* = FALSE). Wenn der Text mit dem Zeichen * beginnt, wird die Leseposition an den Anfang des Textes im markierten Viewer gesetzt (Markierung mit Mauszeiger und F1-Taste). Falls kein Viewer markiert ist, ist *Done* = FALSE. Das Ende des Eingabestroms ist das Ende des Textes, der die Leseposition enthält.

- *OpenText(t, pos)* setzt die Leseposition an die Stelle *pos* im Text *t*. *Done* zeigt den Erfolg der Operation an.

Die folgenden Operationen liefern nur dann korrekte Ergebnisse, wenn vor ihrem Aufruf *Done* = TRUE ist. Alle Operationen außer *Char* überlesen führende Leerzeichen, Tabulatorzeichen oder Zeilenende-Zeichen.

- *Next()* liefert den Symbolcode des nächsten Eingabesymbols (*inval*, *name*, *string*, ...) ohne die Leseposition zu verändern. Wenn es kein nächstes Symbol gibt, liefert *Next* den Wert *inval*.

- *Char(ch)* liefert das Zeichen *ch* an der Leseposition.

- *Int(n)* und *LongInt(n)* liefern die ganzzahlige Konstante *n* an der Leseposition. Wenn die Konstante nicht im Format
  ["-"] digit {digit} | digit {hexDigit} "H"
  ist, wird *Done* auf FALSE gesetzt.

- *Real(n)* liefert die Gleitkommakonstante *n* an der Leseposition. Wenn die Konstante nicht im Format
  ["-"] digit {digit} ["." {digit} ["E" ("+" | "-") digit {digit}]]
  ist, wird *Done* auf FALSE gesetzt.

- *LongReal(n)* liefert die (lange) Gleitkommakonstante *n* an der Leseposition. Wenn die Konstante nicht im Format
  ["-"] digit {digit} "." {digit} [("D" | "E") ("+" | "-") digit {digit}]]
  ist, wird *Done* auf FALSE gesetzt.

- *String(s)* liefert die Zeichenkette *s* an der Leseposition. Wenn die Zeichenkette nicht im Format
  '"' char {char} '"'

ist, wird *Done* auf FALSE gesetzt. Die Zeichenkette im Eingabestrom darf kein Zeichen enthalten, das kleiner als das Leerzeichen ist.

- *Name(s)* liefert den Namen *s* an der Leseposition. Wenn der Name nicht im Format
  letter {letter | digit | "." | "/"}
  ist, wird *Done* auf FALSE gesetzt.

Beispiel

```
VAR i: INTEGER; ch: CHAR; r: REAL; s, n: ARRAY 32 OF CHAR;
...
In.Open;
In.Int(i); In.Char(ch); In.Real(r); InString(s);
IF In.Next() = In.name THEN In.Name(n) END
```

Eingabestrom:

```
123*1.5  "abc"  Mod.Proc
```

Ergebnis:

```
i = 123
ch = "*"
r = 1.5E0
s = "abc"
n = "Mod.Proc"
```

B.2
Modul Out

Modul *Out* bietet Prozeduren zur formatierten Ausgabe von Zeichen, Zahlen und Zeichenketten auf einen Standard-Ausgabestrom. Die hier beschriebene Version von *Out* ist eine Erweiterung der Version aus [Rei91].

```
DEFINITION Out;
   PROCEDURE Open;
   PROCEDURE Close;
   PROCEDURE Char (ch: CHAR);
   PROCEDURE String (str: ARRAY OF CHAR);
   PROCEDURE Int (i, n: LONGINT);
   PROCEDURE Real (x: REAL; n: INTEGER);
   PROCEDURE LongReal (x: LONGREAL; n: INTEGER);
   PROCEDURE Ln;
   PROCEDURE F (str: ARRAY OF CHAR; x: LONGINT);
   PROCEDURE F2 (str: ARRAY OF CHAR; x, y: LONGINT);
   PROCEDURE F3 (str: ARRAY OF CHAR; x, y, z: LONGINT);
   PROCEDURE F4 (str: ARRAY OF CHAR; x, y, z, u: LONGINT);
END Out.
```

Operationen

- *Open* öffnet ein leeres Fenster, in das alle späteren Augaben geschrieben werden. Wenn *Open* nicht aufgerufen wird, gehen alle Ausgaben in den Log-Viewer des Oberon-Systems.

- *Close* schließt das Ausgabefenster. Anschließende Ausgaben gehen in den Log-Viewer.

- *Char(ch)* schreibt das Zeichen *ch* in den Ausgabestrom.

- *String(s)* schreibt das 0X-terminierte Zeichenarry *s* in den Ausgabestrom. Jedes Vorkommen des Zeichens "$" in *s* bewirkt einen Zeilenvorschub.

- *Int(i, n)* schreibt die Zahl *i* in den Ausgabestrom. *n* gibt die Feldweite an. Ist *n* größer als die Anzahl der benötigten Zeichen, wird *i* rechtsbündig in einem Feld von *n* Zeichen ausgerichtet. Ist *n* kleiner als die Anzahl der benötigten Zeichen, wird die Feldweite der Anzahl der benötigten Zeichen angepaßt. Ein positives Vorzeichen wird nicht ausgegeben.

- *Real(x, n)* und *LongReal(x, n)* schreiben die Gleitkommazahl *x* im Exponentialformat in den Ausgabestrom. *n* gibt die Feldweite an und wird wie bei der Operation *Int* behandelt. Ein positives Vorzeichen der Mantisse wird nicht ausgegeben.

- *Ln* sorgt für einen Zeilenvorschub im Ausgabestrom.

- *F(s, x)* schreibt die Zeichenkette *s* in den Ausgabestrom. Das erste Vorkommen des Zeichens "#" in *s* wird durch die Zahl *x* ersetzt. Jeder Vorkommen von "$" wird in einen Zeilenvorschub übersetzt.

- *F2(s, x, y), F3(s, x, y, z), F4(s, x, y, z, u)* verhalten sich wie *F(s, x)*. Vorkommen des Zeichens "#" in *s* werden der Reihe nach durch die Zahlen *x*, *y*, *z*, und *u* ersetzt.

Beispiele

	Ausgabe (Sterne bedeuten Leerzeichen)
Out.Open;	
Out.Int(123, 0);	123
Out.Int(-3, 5);	***-3
Out.Real(1.5, 10);	**1.50E+00
Out.String("first line$second line");	first line second line
Out.F2("x[#] = #$", 1, 2)	x[1] = 2 (*plus Zeilenvorschub*)

B.3
Modul Modules

Modules repräsentiert den Lader des Oberon-Systems. Wir zeigen hier nur einen Auszug seiner Schnittstelle.

```
DEFINITION Modules;

    CONST (*result codes*)
        done = 0; fileNotFound = 1; invalidObjFile = 2; ...

    TYPE
        Module = POINTER TO ModuleDescriptor;
        ModuleDescriptor = RECORD
            link-: Module;                  (*next loaded module *)
            name-: ARRAY 32 OF CHAR;        (*module name *)
            ...
        END;

    VAR
        modules-: Module;           (*list of loaded modules *)
        res-: INTEGER;              (*result code *)

    PROCEDURE ThisMod (name: ARRAY OF CHAR): Module;
    ...

END Modules.
```

Alle geladenen Module sind in einer Liste verkettet, auf deren Anfang die globale Variable *modules* zeigt. Dem *link*-Feld jedes Moduldeskriptors folgend kann man die Liste durchlaufen. Das Feld *name* gibt den Namen des Moduls an. Weitere Felder (die hier nicht gezeigt sind) enthalten z.B. die Adresse des Codes und der Daten des Moduls.

Operationen

- *ThisMod(n)* liefert den Deskriptor des Moduls namens *n*. Falls das Modul noch nicht in der Modulliste stand, wird es geladen. Eventuelle Fehler beim Laden werden in der globalen Variablen *res* angezeigt.

B.4
Modul Types

Modul *Types* verwaltet Laufzeitinformationen über Recordtypen. Es kann dazu verwendet werden, den dynamischen Typ einer Variablen zu bestimmen oder Objekte eines bestimmten dynamischen Typs zu erzeugen.

```
DEFINITION Types;
   IMPORT SYSTEM, Modules;

   TYPE
      Type = POINTER TO TypeDesc;
      TypeDesc = RECORD
         name: ARRAY 32 OF CHAR;
         module: Modules.Module;
      END;

   PROCEDURE TypeOf (o: SYSTEM.PTR): Type;
   PROCEDURE This
      (mod: Modules.Module; name: ARRAY OF CHAR): Type;
   PROCEDURE NewObj (VAR o: SYSTEM.PTR; t: Type);
   PROCEDURE LevelOf (t: Type): INTEGER;
   PROCEDURE BaseOf (t: Type; level: INTEGER): Type;
END Types.
```

Operationen

- *TypeOf(p)* liefert den dynamischen Typ des Record-Objekts, auf das *p* zeigt. Falls *p* auf ein Array-Objekt zeigt, ist das Ergebnis von *TypeOf* undefiniert.

- *This(m, n)* liefert den Deskriptor des Recordtyps namens *n*, der im Modul *m* deklariert ist (oder NIL, falls kein solcher existiert).

- *NewObj(p, t)* liefert einen Zeiger *p* auf ein neues Objekt vom dynamischen Typ *t*.

- *LevelOf(t)* liefert die Erweiterungsstufe des Typs *t*. Ein Typ, der von keinem anderen abgeleitet ist, hat Erweiterungsstufe 0. Ein Typ, der von einem Typ *T* abgeleitet ist, hat die Erweiterungsstufe von *T* plus 1.

- *BaseOf(t, n)* liefert den Basistyp von *t* auf Erweiterungsstufe *n* (oder NIL, falls kein solcher existiert).

B.5
Modul OS

OS ist ein Deckelmodul für andere Oberon-Module (z.B. *Display*, *Files*, *Fonts*, *Input* und *Oberon*), deren Schnittstelle aus Platzgründen hier nicht beschrieben wird. Die Schnittstelle dieser Module kann aus [Rei91] oder [WiG92] entnommen werden. Der Quellcode von *OS* ist auf der Begleit-CD zu diesem Buch enthalten.

```
DEFINITION OS;
    IMPORT Display, Files, Fonts;
    (*Oberon modules which are not explained here*)

    CONST
        right = 0; middle = 1; left = 2;  (*mouse button codes*)
        ticks = 300;  (*OS.Time returns the time in units of 1/ticks seconds*)

    TYPE
        File = Files.File;
        Font = Fonts.Font;
        Message = RECORD END; (*base type for all message records*)
        Object = POINTER TO ObjectDesc;
        Pattern = Display.Pattern;

        Rider = RECORD (Files.Rider) (*read/write position in a file*)
            PROCEDURE (VAR r: Rider) Set (f: Files.File; pos: LONGINT);
            PROCEDURE (VAR r: Rider) Read (VAR x: CHAR);
            PROCEDURE (VAR r: Rider) ReadString
                (VAR s: ARRAY OF CHAR);
            PROCEDURE (VAR r: Rider) ReadInt (VAR x: INTEGER);
            PROCEDURE (VAR r: Rider) ReadLInt (VAR x: LONGINT);
            PROCEDURE (VAR r: Rider) ReadObj (VAR x: Object);
            PROCEDURE (VAR r: Rider) ReadChars
                (VAR x: ARRAY OF CHAR; n: LONGINT);
            PROCEDURE (VAR r: Rider) Write (x: CHAR);
            PROCEDURE (VAR r: Rider) WriteString (s: ARRAY OF CHAR);
            PROCEDURE (VAR r: Rider) WriteInt (x: INTEGER);
            PROCEDURE (VAR r: Rider) WriteLInt (x: LONGINT);
            PROCEDURE (VAR r: Rider) WriteObj (x: Object);
            PROCEDURE (VAR r: Rider) WriteChars
                (VAR x: ARRAY OF CHAR; n: LONGINT);
        END;

        ObjectDesc = RECORD
            PROCEDURE (x: Object) Load (VAR r: Rider);
            PROCEDURE (x: Object) Store (VAR r: Rider);
        END;

    VAR
        Caret-: Pattern;
        screenH-, screenW-: INTEGER; (*screen height and width in pixels*)
```

```
(*screen output*)
   PROCEDURE CopyBlock (sx, sy, w, h, dx, dy: INTEGER);
   PROCEDURE FillBlock (x, y, w, h: INTEGER);
   PROCEDURE EraseBlock (x, y, w, h: INTEGER);
   PROCEDURE InvertBlock (x, y, w, h: INTEGER);
   PROCEDURE DrawPattern (pat: Pattern; x, y: INTEGER);
   PROCEDURE DrawCursor (x, y: INTEGER);
   PROCEDURE FadeCursor;

(*font operations*)
   PROCEDURE DefaultFont (): Font;
   PROCEDURE FontWithName (name: ARRAY OF CHAR): Font;
   PROCEDURE GetCharMetric (f: Font; ch: CHAR;
      VAR dx, x, y, w, h: INTEGER; VAR pat: Pattern);

(*keyboard and mouse input*)
   PROCEDURE AvailChars (): INTEGER;
   PROCEDURE ReadKey (VAR ch: CHAR);
   PROCEDURE GetMouse (VAR buttons: SET; VAR x, y: INTEGER);

(*disk input/output*)
   PROCEDURE NewFile (name: ARRAY OF CHAR): File;
   PROCEDURE OldFile (name: ARRAY OF CHAR): File;
   PROCEDURE Register (f: File);
   PROCEDURE InitRider (VAR r: Rider);

(*miscellaneous*)
   PROCEDURE Move (VAR fromBuf: ARRAY OF CHAR; from: LONGINT;
      VAR toBuf: ARRAY OF CHAR; to, n: LONGINT);
   PROCEDURE Time (): LONGINT;
   PROCEDURE Call (command: ARRAY OF CHAR);

END OS.
```

Bildschirmausgabeoperationen

- *CopyBlock (sx, sy, w, h, dx, dh)* kopiert den rechteckigen Bildschirmbereich *(sx, sy, w, h)* nach *(dx, dy, w, h)*.
- *FillBlock (x, y, w, h)* füllt den rechteckigen Bildschirmbereich *(x, y, w, h)* schwarz.
- *EraseBlock (x, y, w, h)* löscht den rechteckigen Bildschirmbereich *(x, y, w, h)*.
- *InvertBlock (x, y, w, h)* invertiert den rechteckigen Bildschirmbereich *(x, y, w, h)*.
- *DrawPattern (pat, x, y)* kopiert das rechteckige Muster *pat* an die Bildschirmposition, deren linke untere Ecke *(x, y)* ist.
- *DrawCursor (x, y)* bewegt den Mauspfeil von seiner momentanen Position auf dem Bildschirm zur Position *(x, y)*.
- *FadeCursor* blendet den Mauspfeil aus.

Font-Operationen

- *f := DefaultFont ()* liefert in *f* den Standardfont.
- *f := FontWithName (name)* liefert in *f* den Font namens *name*.
- *GetCharMetric (fnt, ch, dx, x, y, w, h, pat)* liefert die Zeichenmetrik und das Muster *pat* des Zeichens *ch* im Font *fnt*. Die Bedeutung von *dx*, *x*, *y*, *w* und *h* ist Abb. 12.22 zu entnehmen.

Eingabeoperationen für Maus und Tastatur

- *n := AvailChars ()* liefert in *n* die Anzahl der Zeichen im Tastaturpuffer.
- *ReadKey (ch)* liest und entfernt das nächste Zeichen *ch* aus dem Tastaturpuffer. Wenn dieser leer ist, blockiert die Prozedur, bis ein Zeichen eingetippt wird.
- *GetMouse (b, x, y)* liefert die Mauskoordinaten *(x, y)* relativ zur linken unteren Ecke des Bildschirms sowie die Menge *b* der gedrückten Mausknöpfe (0 = right, 1 = middle, 2 = left).

Datei-Operationen

- *f := NewFile (name)* legt eine neue (temporäre) Datei *f* namens *name* an und öffnet sie.
- *f := OldFile (name)* öffnet die Datei *f* namens *name*. Wenn keine solche Datei existiert, ist *f* = NIL.
- *Register (f)* wandelt die mit *NewFile* angelegte temporäre Datei *f* in eine permanente Datei um.
- *InitRider (r)* initialisiert den Rider *r*.

Methoden der Klasse Rider

- *r.Set (f, pos)* setzt den Rider *r* auf die Position *pos* in der Datei *f*.
- *r.Read (ch)* liest ein Zeichen *ch* vom Rider *r*.
- *r.ReadInt (x)* liest eine Integer-Zahl *x* vom Rider *r*.
- *r.ReadLInt (x)* liest eine Longinteger-Zahl *x* vom Rider *r*.
- *r.ReadString (s)* liest ein Zeichenarray *s* vom Rider *r*.
- *r.ReadChars (buf, len)* liest *len* Zeichen vom Rider *r* in den Puffer *buf*.
- *r.ReadObj (obj)* liest und erzeugt ein beliebiges mit *WriteObj* ausgegebenes Objekt und liefert es in *obj* (siehe Kapitel 9.4.6).
- *r.Write (ch)* schreibt ein Zeichen *ch* auf den Rider *r*.
- *r.WriteInt (x)* schreibt eine Integer-Zahl *x* auf den Rider *r*.
- *r.WriteLInt (x)* schreibt eine Longinteger-Zahl *x* auf den Rider *r*.
- *r.WriteString (s)* schreibt das Zeichenarray *s* auf den Rider *r*.
- *r.WriteChars (buf, len)* schreibt *len* Zeichen vom Puffer *buf* auf den Rider *r*.
- *r.WriteObj (obj)* schreibt auf den Rider *r* ein beliebiges Objekt *obj* (siehe Kapitel 9.4.6).

Andere Operationen

- *Move (buf0, pos0, buf1, pos1, len)* kopiert *len* Bytes von *buf0[pos0]* nach *buf1[pos1]*.
- *t := Time ()* liefert die seit Systemstart verstrichene Zeit in Einheiten von 1/*ticks* Sekunden (*ticks* ist eine Konstante des Moduls *OS*).
- *Call (cmd)* aktiviert das Kommando *cmd* und lädt dabei das Modul, das das Kommando enthält, falls dieses noch nicht geladen war.

C Glossar

Abstrakte Datenstruktur (ADS)
Eine Datenstruktur, von der man nur die Schnittstelle (die zu ihrer Benutzung nötigen Operationen), nicht aber ihre Implementierung kennt. Kann in Form eines Moduls implementiert werden.

Abstrakte Klasse
Eine Klasse, die zur Definition einer gemeinsamen Schnittstelle für zukünftige Unterklassen dient. Abstrakte Klassen besitzen normalerweise keine Attribute. Ihre Methoden sind abstrakt (ohne Implementierung). Es gibt keine Objekte einer abstrakten Klasse.

Abstrakte Methode
Eine Methode, zu der es keine Implementierung gibt. Sie definiert lediglich die Schnittstelle gleichnamiger Methoden in Unterklassen und muß dort überschrieben werden.

Abstrakter Datentyp (ADT)
Ein Datentyp bestehend aus Daten und Operationen. Die Implementierung der Daten ist versteckt. Zugriff auf die Daten ist nur mit Hilfe der definierten Operationen möglich. Es können mehrere Variablen eines abstrakten Datentyps deklariert werden.

Adapter (Wrapper)
Entwurfsmuster. Ein Adapter macht eine bestehende Klasse C zu einer fremden Klassenfamilie kompatibel. Der Adapter ist Mitglied der Klassenfamilie und leitet alle Meldungen an das C-Objekt weiter.

Aggregation
Zusammenfassung mehrerer Objekte als Teile (Attribute) eines neuen Objekts.

Application Framework
Ein Framework, das die gemeinsamen Teile einer Menge ähnlicher Applikationen enthält (z.B. Arbeiten mit Fenstern und Menüs, Öffnen und Schließen von Dokumenten, etc). Es kann durch Vererbung und Zusammenstecken benutzerspezifischer Objekte zu verschiedenen Applikationen ausgebaut werden.

Attribut (Instanzvariable)
Datenkomponente einer Klasse.

Basisklasse
Siehe Oberklasse.

Benutzt-Beziehung
Beziehung zwischen Objekten. Ein Objekt a benutzt ein Objekt b, wenn es auf seine Attribute zugreift, oder ihm Meldungen schickt.

Beobachter (Observer)
Entwurfsmuster. Ein Objekt, das benachrichtigt wird, wenn sich ein bestimmtes anderes Objekt ändert. Mehrere Observer können auf dasselbe Objekt angesetzt werden. Beispiel: Sichten als Observers eines Datenmodells (Siehe MVC-Schema, Change Propagation).

Beziehungen zwischen Objekten und Klassen
Ist-Beziehung, Hat-Beziehung, Benutzt-Beziehung

Browser
Werkzeug zur Anzeige von Informationen über ein objektorientiertes System. Z.B. Anzeige von Klassenhierarchien, Klassenschnittstellen oder Klassenbeziehungen.

Change Propagation
Die Benachrichtigung abhängiger Objekte, wenn sich ein Objekt ändert. Siehe auch Observer.

Composite
Entwurfsmuster. Ein aus mehreren Teilen zusammengesetztes Objekt, das selbst wie ein Teil aussieht und daher wieder zu größeren Objekten zusammengesetzt werden kann. Das Composite-Objekt und die Teilobjekte müssen die gleiche Schnittstelle aufweisen, also von der gleichen abstrakten Klasse abgeleitet sein.

Container
Ein Objekt, das andere Objekte enthält. Die Ordnung und Implementierung der enthaltenen Objekte führt zu verschiedenen Container-Arten (Set, Bag, List, SortedList, HashList, ...).

CRC-Karte (Class-Responsibilities-Collaborators-Karte)
Einfaches Hilfsmittel zum Sammeln von Klassen und ihrer Spezifikation. Auf eine Karte werden der Name der Klasse, ihre Aufgaben (responsibilities) und ihre Partner (collaborators) notiert. Die Karten der Klassen eines Systems können übersichtlich auf einem Tisch ausgebreitet werden.

C++
Hybride objektorientierte Sprache, die auf C aufbaut und folgende Eigenschaften hinzufügt: Klassen mit mehrfacher Vererbung; statische und dynamische Bindung; Konstruktoren und Destruktoren; Templates; Overloading; Ausnahmebehandlung. Es fehlt vor allem Garbage Collection und eine Typenprüfung, die nicht umgangen werden kann.

Datenabstraktion
Das Weglassen von Implementierungsdetails in der Schnittstelle eines Bausteins (Klasse, Modul, ADT). Die Benutzung des Bausteins wird über wenige Prozeduren ermöglicht.

Datenkapsel
Siehe Abstrakte Datenstruktur.

Decorator
Entwurfsmuster. Ein Objekt, mit dem man zur Laufzeit Eigenschaften zu einem anderen Objekt hinzufügen kann, indem man den Decorator vor das andere Objekt schaltet. Ein Decorator d und ein dekoriertes Objekt o müssen die gleiche Schnittstelle haben. Meldungen an d werden nach ihrer Behandlung an o weitergeleitet (Siehe Forwarding). Für Klienten sieht d wie o aus.

Deep Copy
Das Kopieren eines Objekts und aller von ihm direkt oder indirekt referenzierten Objekte (d.h. Kopieren des gesamten Graphen, der an einem Objekt hängt). Gegenteil: Shallow Copy.

Delegation
Weiterleiten einer Meldung an ein anderes Objekt. Unterschied zu Forwarding: Wenn ein Objekt x eine Meldung an ein Objekt y delegiert und y schickt sich anschließend selbst eine Meldung, so beginnt die Methodensuche wieder bei x (d.h. der ursprünglich Empfänger wird beibehalten), während bei Forwarding die Methodensuche bei y beginnt (d.h. der Empfänger ändert sich). Delegation ist eine Alternative zur Vererbung, die es sogar erlaubt, die Delegations- (Vererbungs-) hierarchie zur Laufzeit zu ändern.

Design Pattern
Siehe Entwurfsmuster.

Destruktor
C++-Begriff. Eine Methode, die automatisch aufgerufen wird, wenn ein Objekt freigegeben wird. In einem Destruktor können Abschlußarbeiten erledigt werden (z.B. Freigeben von Attributen).

Dynamische Bindung (Late Binding)
Meldungen werden normalerweise dynamisch gebunden, d.h. es wird erst zur Laufzeit bestimmt, welche Methode durch eine Meldung ausgelöst wird. Eine Meldung M an ein Objekt o führt zum Aufruf derjenigen Methode M, die zum dynamischen Typ von o gehört.

Dynamische Typprüfung
Dynamische Typprüfung liegt vor, wenn Variablen ohne Typ deklariert werden und daher Objekte beliebigen Typs enthalten können. Typprüfungen finden erst zur Laufzeit statt. Zum Beispiel wird zur Laufzeit geprüft, ob eine Operation mit den dynamischen Typen der Operanden verträglich ist. Beispiele für Sprachen mit dynamischer Typprüfung sind Smalltalk und Self.

Dynamischer Typ (Laufzeittyp)
Der dynamische Typ einer Variablen ist der Typ des Objekts, das diese Variable zur Laufzeit enthält (bzw. das sie referenziert). Er kann eine Erweiterung des statischen Typs dieser Variablen sein.

Dynamisches Modell
Zur Analyse eines Systems vorgenommene Modellierung seiner Ereignisse und Zustände. Als Beschreibungsmittel werden Ereignisdiagramme und Zustandsübergangsdiagramme verwendet. Teil der OMT- und UML-Methode.

Early Binding
Siehe Statische Bindung.

ECOOP
European Conference on Object-Oriented Programming.

Eiffel
Objektorientierte Sprache: Klassen mit mehrfacher Vererbung; dynamische Bindung; Garbage Collection; Konstruktoren; Ausnahmebehandlung; strenge statische Typenprüfung; keine Module.

Empfänger (Receiver)
Als Empfänger einer Meldung bezeichnet man das Objekt, an das die Meldung gesendet wird (das Objekt, auf das sich die Operation bezieht). Der Empfänger wird als Parameter an die aufgerufene Methode übergeben. In den meisten Sprachen hat der Empfänger in der aufgerufenen Methode einen vordeklarierten Namen (self, this, ...) und wird versteckt übergeben. In Oberon-2 ist die Übergabe explizit.

Entwurfsmuster (Design Pattern)
Ein Standardproblem und seine schematische Lösung mittels kooperierender Klassen.

Interaktionsdiagramm (Ereignisdiagramm)
Grafische Darstellung des Meldungsflusses zwischen Objekten. Objekte werden durch senkrechte Linien dargestellt, Meldungen durch Pfeile zwischen der Linie des sendenden und des empfangenden Objekts.

Erweiterung
Siehe Unterklasse.

Fabrik (auch abstrakte Fabrik)
Entwurfsmuster. Eine Fabrik erzeugt und liefert Objekte mit einer vorgegebenen Schnittstelle aber mit verschiedenen dynamischen Typen. Durch Austauschen eines Fabrik-Objekts kann mit einem Schlag der dynamische Typ aller durch das Fabrik-Objekt erzeugten Objekte geändert werden.

Forwarding
Weiterleiten einer Meldung an ein anderes Objekt. Unterschied zu Delegation (siehe dort).

Framework (Gerüst)
Eine Menge erweiterbarer Klassen, die zusammenarbeiten, um den gemeinsamen Kern ähnlicher Systeme zu realisieren. Durch Erweitern der Framework-Klassen sowie durch Hinzufügen von systemspezifischen Klassen, kann ein Framework zu verschiedenen Endsystemen ausgebaut werden. Ein Framework ist oft bereits ablauffähig und ruft an gewissen Stellen Methoden auf, die der Programmierer zur Verfügung stellen muß.

Friend
C++-Begriff. Eine Prozedur oder Klasse ist Freund einer anderen Klasse X, wenn sie Zugriff auf die privaten (und daher verborgenen) Attribute von X hat. Bei der Deklaration von X müssen alle ihre Freunde angegeben werden. In Oberon-2 haben alle im gleichen Modul deklarierten Klassen automatisch gegenseitigen Zugriff zu ihren Attributen. Sie sind daher implizit Freunde.

Garbage Collection
Automatische Speicherbereinigung. Das Laufzeitsystem sammelt automatisch den Speicherplatz aller nicht mehr referenzierten Objekte ein. Bequem und sicher, besonders in objektorientierten Programmen, in denen Objekte oft von privaten Attributen referenziert werden und daher von Klienten, die auf diese Attribute keinen Zugriff haben, gar nicht freigegeben werden können.

Geheimnisprinzip
Siehe Information Hiding.

Generalisierung
Zusammenfassung ähnlicher Dinge unter einem gemeinsamen Oberbegriff. Im objektorientierten Sinn das Zusammenfassen mehrerer Klassen unter einer gemeinsamen Oberklasse.

Generische Klasse (Template)
Eine Klasse, in der die Typen einiger Attribute oder Methodenparameter noch offengelassen wurden. Bei der Verwendung der Klasse in einer Deklaration werden die fehlenden Typen als Parameter angegeben, wodurch aus einer generischen Klasse verschiedene konkrete Klassen erzeugt werden können.

Gerüst
Siehe Framework.

Hat-Beziehung
Beziehung zwischen Objekten. Ein Objekt x hat ein Objekt y, wenn y ein Attribut von x ist und man sagen kann: "x besteht aus y".

Hybride objektorientierte Sprache
Eine Sprache, in der es neben Klassen auch andere Datentypen (z.B. Integer) gibt und neben Meldungen auch andere Operationen (z.B. Prozeduren, arithmetische Operatoren, ...). Vorteil: größere Effizienz; Nachteil: uneinheitliche Behandlung von Klassen und anderen Typen. Beispiele: C++, Object-Pascal, Java, Oberon-2.

Information Hiding (Geheimnisprinzip)
Ein von David Parnas postuliertes Prinzip, nach dem ein Baustein (Klasse, Modul, ADT) die Implementierung seiner Daten vor Klienten verbergen sollte. Dies macht die Benutzung des Bausteins für Klienten einfacher und erlaubt es, die Implementierung der Daten zu ändern, ohne Klienten ändern zu müssen. Information Hiding bietet auch Schutz vor ungewollter Zerstörung privater Daten.

Inheritance
Siehe Vererbung.

Instanz
Objekt einer Klasse.

Instanzvariable
Siehe Attribut.

Ist-Beziehung
Beziehung zwischen Objekten. Ein Objekt einer Klasse U ist auch ein Objekt einer Klasse O, wenn O eine Oberklasse von U ist.

Iterator
Eine Operation zum sequentiellen Durchlaufen einer Menge von Objekten, wobei den Klienten nicht bekannt ist, wie die Objekte der Menge miteinander verknüpft sind.

Klasse
Abstrakter Datentyp mit Vererbungsmöglichkeit. Die Daten einer Klasse nennt man Attribute, ihre Operationen Methoden. Die Implementierung der Attribute ist meist verborgen (Information Hiding), sodaß alle Zugriffe mittels Methoden erfolgen. Methodenaufrufe (Meldungen) werden meist dynamisch gebunden. Manchmal wird zwischen Klasse und Typ unterschieden: Ein Typ spezifiziert die Schnittstelle einer Klasse; eine Klasse ist die Implementierung eines Typs.

Klassenbibliothek
Sammlung relativ unzusammenhängender Klassen als wiederverwendbare und erweiterbare Bausteine. Im Gegensatz dazu ist ein Framework eine Sammlung zusammenarbeitender Klassen.

Klassenbrowser
Werkzeug zur Darstellung der Klassenhierarchie eines Systems sowie der Klassenschnittstellen.

Klassenhierarchie
Die hierarchische Darstellung der Vererbungs-Beziehungen zwischen Klassen. Meist als Baum (bei mehrfacher Vererbung als azyklischer Graph) oder als Mengendiagramm dargestellt.

Klasseninvariante
Eine boolesche Aussage über die Attribute einer Klasse, die vor und nach der Ausführung jeder Methode der Klasse gilt.

Konkrete Klasse
Eine Klasse, in der alle Methoden eine Implementierung besitzen. Gegenteil: Abstrakte Klasse.

Konstruktor (Constructor)
Eine Operation zum Erzeugen und Initialisieren von Objekten einer bestimmten Klasse. In C++ und Java als Sprachkonstrukt vorhanden.

Kontrakt
Ein Übereinkommen zwischen einer Klasse und ihren Klienten, das mittels Vor- und Nachbedingungen von Methoden spezifiziert wird. Wenn ein Klient vor dem Aufruf einer Methode ihre Vorbedingung garantiert, dann garantiert die Methode, daß bei ihrer Rückkehr die Nachbedingung gilt. Beim Überschreiben einer Methode in einer Unterklasse darf die Vorbedingung abgeschwächt und die Nachbedingung verstärkt werden, d.h. Unterklassen sind korrekt, wenn sie gleich viel oder weniger erwarten und gleich viel oder mehr garantieren als ihre Oberklasse.

Kontravarianz
Parametertypen überschriebener Methoden ändern sich entgegen der Klassenhierarchie. Kontravarianz liegt vor, wenn eine Methode einen Parameter vom Typ B hat und dieser beim Überschreiben der Methode in einen Parameter vom Typ A geändert wird, wobei A eine Oberklasse von B ist. Für Eingangsparameter ist Kontravarianz typsicher, da A eine schwächere

Vorbedingung darstellt als B (Siehe Kontrakt, Kovarianz). Allerdings ist Kontravarianz kaum nützlich.

Kovarianz
Parametertypen überschriebener Methoden ändern sich in Richtung der Klassenhierarchie. Kovarianz liegt vor, wenn eine Methode einen Parameter vom Typ A hat und dieser beim Überschreiben der Methode in einen Parameter vom Typ B geändert wird, wobei A eine Oberklasse von B ist. Für Ausgangsparameter ist Kovarianz typsicher, da B eine stärkere Nachbedingung darstellt als A (Siehe Kontrakt, Kontravarianz).

Late Binding
Siehe Dynamische Bindung.

Laufzeittyp
Siehe Dynamischer Typ.

Mehrfache Vererbung (Multiple Inheritance)
Die Ableitung einer Klasse U aus mehreren Oberklassen O1 ... On. U erbt alle Komponenten der Oberklassen. Enthalten Oberklassen gleichnamige Komponenten, so kommt es zu einem Namenskonflikt (der Name ist in der Unterklasse nicht mehr eindeutig). Namenskonflikte können durch Umbenennungen oder durch Qualifikation mit dem Namen der Oberklasse aufgelöst werden.

Meldung (Message)
Ein Auftrag an ein Objekt (Empfänger), eine bestimmte Operation auszuführen. Meldungen werden üblicherweise dynamisch gebunden, d.h. es wird diejenige Methode gleichen Namens aufgerufen, die zum dynamischen Typ des Empfängers gehört.

Meldungsobjekt
Entwurfsmuster. Eine Meldung, die als Objekt (d.h. als Daten) aufgefaßt und einem anderen Objekt zur Interpretation übergeben wird. Die Interpretation erfolgt zur Laufzeit. Flexibler aber auch umständlicher als der Aufruf von Methoden.

Member
C++-Begriff für Attribute oder Methoden einer Klasse.

Message
Siehe Meldung.

Methode (Typgebundene Prozedur)
Eine Operation auf Objekte einer Klasse. Sie ist Bestandteil dieser Klasse und wird durch Senden einer Meldung an das Objekt aufgerufen.

Methodentabelle
Eine Tabelle mit den Adressen aller Methoden einer Klasse. Jedes Objekt besitzt einen verborgenen Zeiger auf die Methodentabelle seiner Klasse, die mit Methodennummern indiziert wird.

Mixin-Klasse
Eine Klasse, die mittels mehrfacher Vererbung zu einer anderen Klasse "dazugemischt" wird, um ihre Eigenschaften auf die andere Klasse zu übertragen.

Multiple Inheritance
Siehe Mehrfache Vererbung.

MVC-Schema (Model-View-Controller-Schema)
Häufiges Muster in interaktiven Programmen, das auch oft als Framework vorliegt: Auf bestimmte Daten (Model) gibt es mehrere Sichten (Views), die mittels Eingabekomponenten (Controllers) manipuliert werden. Eine Eingabe an einen Controller bewirkt eine Änderung des Modells, das daraufhin alle seine Sichten benachrichtigt und auffordert, sich nachzuführen. Dadurch wird garantiert, daß alle Sichten konsistent sind, d.h. denselben Zustand des Modells zeigen.

Oberklasse (Basisklasse, Superclass)
Eine Klasse O ist Oberklasse einer Klasse U, wenn U mittels Vererbung von O abgeleitet ist. Die Oberklassen-Beziehung ist transitiv: Oberklassen von O sind auch Oberklassen von U.

Oberon-2
Hybride objektorientierte Sprache auf der Basis von Modula-2: Klassen mit einfacher Vererbung, dynamische Bindung, strenge statische Typenprüfung, Garbage Collection, Laufzeit-Typtest.

Objekt
Ein Exemplar einer Klasse, d.h. ein Wert (oder Wertetupel), dessen Struktur und Verhalten durch die Klasse festgelegt wird.

Objektbasiert
Eine Sprache heißt objektbasiert, wenn sie abstrakte Datentypen aber keine Vererbung und dynamische Bindung anbietet.

Objektorientiert
Eine Sprache heißt objektorientiert, wenn sie abstrakte Datentypen, Vererbung und dynamische Bindung anbietet.

Objektorientierte Analyse (OOA)
Der Vorgang, die zur Modellierung eines Systems benötigten Klassen, ihre Aufgaben und ihr Zusammenspiel herauszufinden. Ausgangspunkt ist meist eine Anforderungsdefinition. Ergebnis ist eine Menge von Klassen, z.B. als CRC-Karten oder als Klassendiagramm. Es wird noch keine Rücksicht auf Implementierungsdetails oder auf eine bestimmte Programmiersprache genommen.

Objektorientierter Entwurf (Object-Oriented Design, OOD)
Der Entwurf der für ein System benötigten Klassen und ihrer Schnittstellen, sowie ihre Zuordnung zu Modulen und Prozessen. Oft ist die Grenze zwischen objektorientierter Analyse und objektorientiertem Entwurf fließend.

Objektmodell
Grafische Beschreibung aller in einem System vorkommenden Klassen samt Attributen und Methodenschnittstellen sowie ihrer Beziehungen zueinander (Ist-Beziehung, Hat-Beziehung, Benutzt-Beziehung). Das Objektmodell ist eine statische Beschreibung, aus der keine Abläufe ersichtlich sind.

Objektreferenz
Zeiger auf ein Objekt. Die meisten objektorientierten Programme arbeiten mit Objektreferenzen, da Polymorphismus und dynamische Bindung nur dann funktionieren, wenn Variablen Referenzen anstatt der eigentlichen Objekte enthalten.

Observer
siehe Beobachter.

OMT
Object Modeling Technique. Verbreitete Methode zur objektorientierten Analyse und zum objektorientierten Entwurf. Definiert vor allem Notationen zur Beschreibung des Objektmodells, des dynamischen Modells und des funktionalen Modells eines Systems.

OOPL
Object-Oriented Programming Language.

OOPSLA
Conference on Object-Oriented Programming: Systems, Languages and Applications.

Overloading
Deklaration mehrerer Methoden gleichen Namens in derselben Klasse. Beim Aufruf erfolgt die Auswahl der Methode auf Grund der Anzahl und Typen der aktuellen Parameter. In C++ können auch Operatoren (+, -, ...) als Methoden deklariert werden. Da diese Operatoren die gleichnamigen Standardoperatoren überschreiben, spricht man von Operator Overloading.

Overriding
Siehe Überschreiben.

Pattern
Siehe Entwurfsmuster.

Persistentes Objekt
Ein Objekt, das auch dann existiert, wenn kein Programm läuft, von dem es benutzt wird. Persistente Objekte werden üblicherweise auf einer Datei zwischengespeichert, bis sie von einem anderen Programm wieder in den Speicher geholt und reaktiviert werden.

Polymorphismus
- Datenpolymorphismus: Eine Variable ist polymorph, wenn sie Objekte verschiedenen Typs enthalten kann.
- Operationspolymorphismus: Eine Operation ist polymorph, wenn sie auf Operanden verschiedenen Typs anwendbar ist.

Protokoll
Schnittstelle einer Klasse, d.h. alle für Klienten sichtbaren Attribute und Methoden.

Prototyp
- Entwurfsmuster. Ein Musterobjekt, das dazu dient, Kopien von sich anzufertigen. Statt ein Objekt einer Klasse zu erzeugen wird eine Kopie eines (bereits initialisierten) Prototyps erzeugt.

- Konstrukt bestimmter objektorientierter Programmiersprachen (z.B. Self). Diese Sprachen besitzen meist keine Klassen. Anstatt Vererbung wird Delegation benutzt.

Receiver
Siehe Empfänger.

Rein objektorientierte Sprache (Purely Object-Oriented Language)
Eine Sprache, in der alle Datentypen Klassen und alle Operationen Meldungen sind. Vorteil: einheitliche Behandlung aller Programmbausteine; Nachteil: geringere Effizienz. Typische Vertreter: Smalltalk. Gegenteil: hybride objektorientierte Sprache.

Schablonenmethode (Template Method)
Entwurfsmuster. Eine Methode, die abstrakte Methoden aufruft. Damit läßt sich das Skelett eines Algorithmus fixieren, wobei Details des Algorithmus durch Überschreiben der abstrakten Methoden implementiert werden.

Schnittstelle einer Klasse
Alle für andere Klassen sichtbaren Attribute (samt Typen) und Methoden (samt Parameterlisten).

Shallow Copy
Das Kopieren eines Objekts, wobei die von ihm referenzierten Objekte nicht kopiert werden. Gegenteil: Deep Copy.

Simula
Erste objektorientierte Sprache (1967). Klassen mit einfacher Vererbung; dynamische Bindung; Objekte als Koroutinen; Garbage Collection.

Smalltalk
Rein objektorientierte Sprache. Eine der ersten, konsequentesten und bekanntesten objektorientierten Sprachen. Klassen mit einfacher Vererbung; dynamische Bindung; dynamische Typenprüfung; Metaklassen; Blöcke; Garbage Collection; umfangreiche Klassenbibliothek und bequeme Programmierwerkzeuge.

Statische Bindung (Early Binding)
Prozeduraufrufe werden statisch gebunden, d.h. es ist zur Übersetzungszeit (also statisch) bekannt, zu welcher Prozedur ein Aufruf führt. In manchen Sprachen (z.B. C++) können auch Meldungen statisch gebunden werden, was zu größerer Effizienz aber zu geringerer Flexibilität führt, da solche Meldungen nicht mittels Polymorphismus arbeiten können.

Statische Typprüfung
Statische Typprüfung liegt vor, wenn Variablen mit einem Typ deklariert werden und der Compiler prüft, ob die Variablen entsprechend ihres Typs korrekt verwendet werden. Beispiele für Sprachen mit statischer Typprüfung sind C++, Java, Eiffel oder Oberon-2.

Statischer Typ
Der statische Typ einer Variablen ist der Typ mit dem sie deklariert wurde.

Subtyp
Siehe Unterklasse.

Supercall
Führt eine Methode einer Klasse C einen Supercall aus, so wird die gleichnamige Methode aus der Oberklasse von C aufgerufen. Dient zur Aktivierung des geerbten, aber überschriebenen Verhaltens.

Superclass
Siehe Oberklasse.

Supertyp
Siehe Oberklasse.

Template
Siehe Generische Klasse.

Template Method
Siehe Schablonenmethode.

Twin
Siehe Zwillingsklasse.

Typ
Beschreibung der Struktur und der Operationen einer Menge von Objekten. Im allgemeinen gleichgesetzt mit Klasse. Manchmal wird unterschieden: ein Typ spezifiziert die Schnittstelle einer Klasse; eine Klasse ist die Implementierung eines Typs.

Typdeskriptor
Metainformation, die zur Laufzeit über jede Klasse gespeichert wird. Jedes Objekt hat einen versteckten Zeiger auf seinen Typdeskriptor. Der Typdeskriptor enthält üblicherweise den Typnamen, die Namen, Typen und Adressen der Attribute sowie die Methodentabelle.

Type Cast
Umwandlung des statischen Typs einer Variablen in einen anderen Typ. Dabei wird lediglich das Bitmuster der Variablen anders interpretiert. Es findet keine Prüfung statt, ob die Umwandlung erlaubt ist. Daher sind Type Casts sehr gefährlich und nicht zu empfehlen.

Type Guard
Oberon-2-Begriff. Type Cast mit Laufzeittypprüfung. Der statische Typ einer Variablen v darf in den Typ T umgewandelt werden, wenn der dynamische Typ von v T oder ein Untertyp von T ist.

Typerweiterung
Siehe Vererbung.

Typgebundene Prozedur
Oberon-2-Begriff für Methode.

Typtest
Laufzeitprüfung, ob der dynamische Typ einer Variablen ein bestimmter Typ T ist.

Typzusicherung
Siehe Type Guard.

Überschreiben (Overriding)
Neudeklaration einer geerbten Methode in einer Unterklasse. Dient zur Anpassung geerbten Verhaltens oder zur Implementierung abstrakter

Methoden in Unterklassen. Die überschreibende und die geerbte Methode müssen gleichen Namen und gleiche Parameterlisten haben. Manche Sprachen erlauben auch kovariante oder kontravariante Parameterlisten. Die überschreibende Methode verdeckt die geerbte Methode, d.h. eine entsprechende Meldung an ein Objekt der Unterklasse führt zum Aufruf der neu deklarierten Methode. Die geerbte Methode kann von der neu deklarierten Methode mittels Supercall aufgerufen werden.

UML
Unified Modeling Language. Methode zur objektorientierten Analyse und zum objektorientierten Entwurf. Entstanden als Vereinigung dreier anderer Methoden (OMT, Booch-Methode, Use Case Driven Design). Definiert vor allem Notationen zur Beschreibung des Objektmodells, des dynamischen Modells und die Analyse eines Systems mittels Use Cases.

Unterklasse (Subclass, Untertyp, Erweiterung)
Eine von einer Klasse O (Oberklasse) mittels Vererbung abgeleitete Klasse U. U erbt alle Attribute und Methoden von O und deren Oberklassen und kann weitere Attribute und Methoden hinzufügen. U ist eine Erweiterung (Spezialisierung) von O, daher ist jedes U-Objekt auch ein O-Objekt und kann überall dort verwendet werden, wo ein O-Objekt erwartet wird (Kompatibilität). Die Unterklassen-Beziehung ist transitiv: Unterklassen von U sind auch Unterklassen von O.

Untertyp
Siehe Unterklasse.

Vererbung (Inheritance, Typerweiterung)
Beziehung zwischen Klassen. Eine Unterklasse U erbt alle Komponenten ihrer Oberklasse O, d.h. diese Komponenten sind in U vorhanden, wie wenn sie dort deklariert worden wären. Vererbung stellt auch eine Ist-Beziehung zwischen Objekten dar, indem jedes U-Objekt auch ein O-Objekt ist (Kompatibilität).

Virtuelle Methode
C++-Begriff. Methode, die mittels dynamischer Bindung aufgerufen werden kann. Im Gegensatz zu C++ sind in den meisten objektorientierten Sprachen alle Methoden virtuell.

Wrapper
Siehe Adapter.

Zustandsübergangsdiagramm
Ein Graph, bei dem die Knoten Zustände und die Kanten Zustandsübergänge darstellen. Die Beschriftung der Kanten gibt das Ereignis an, bei dem der Zustandsübergang stattfindet. Teil der OMT- und der UML-Methode.

Zwillingsklasse (Twin)
Entwurfsmuster. Zur Implementierung mehrfacher Vererbung in Sprachen, die dieses Konstrukt nicht anbieten. Wenn eine Klasse C von zwei Klassen A und B erben soll, so wird sie in Zwillinge CA und CB zerlegt, wobei CA von A und CB von B erbt. CA leitet B-Meldungen an CB weiter und umgekehrt.

D Bezugsquellen

Diesem Buch liegt eine CD mit folgendem Inhalt bei:

CD-Inhalt

- Die Quellprogramme der Fallstudie Oberon0 aus Kapitel 12.
- Das ETH-Oberon-System V4 für die Betriebssysteme Windows 95/NT, MacOS (auf Power Macintosh) und Linux. Darin sind der Oberon-2-Compiler, diverse Editoren, Browser und sonstige Werkzeuge enthalten sowie die Oberon Laufzeitumgebung mit Garbage Collector, Kommandoaktivierung und dynamischem Laden von Modulen. Die CD enthält sowohl die lauffähige Version des Oberon-Systems als auch sämtliche Quellprogramme des Betriebssystems, des Compilers und der Werkzeuge in der Version der Universität Linz.

Sowohl die Fallstudie als auch das Oberon-System können auch über das Internet bezogen werden

Bezug über das Internet

- Fallstudie
  ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon/Oberon0.Cod
- ETH-Oberon-System V4
  ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon/
  ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/

Neben dem an der ETH Zürich entwickelten Oberon-System gibt es noch eine Reihe anderer Oberon-Implementierungen, die teilweise kommerziell vertrieben werden, teilweise aber auch kostenlos über das Internet erhältlich sind. Auch von den kommerziellen Systemen gibt es meist eine kostenlose Version für Ausbildungszwecke.

Andere Oberon-Systeme

- *BlackBox Component Builder*
  Hinter diesem Namen verbirgt sich ein Oberon-System mit Oberon-2-Compiler und Entwicklungsumgebung ähnlich wie ETH Oberon V4. Das System wird von der Firma Oberon microsystems (*http://www.oberon.ch*) vertrieben. Es existiert in Versionen für Windows 95/NT und MacOS. Im Unterschied zu ETH Oberon V4 paßt es sich an die Oberfläche des Gastbetriebssystems an. Blackbox Component Builder wird von Oberon microsystems kommerziell vertrieben. Es gibt allerdings eine kostenlose Version für Ausbildungszwecke.

- *Pow!*
  Pow! ist eine Oberon-Entwicklungsumgebung unter Windows, die aus Oberon-2 Programmen Windows-EXE-Dateien erzeugt. Sie wurde am Forschungsinstitut für Mikroprozessortechnik der Universität Linz entwickelt. Nähere Informationen zu Pow! findet man unter *http://www.fim.uni-linz.ac.at/pow/pow.htm*.

- *ETH Oberon System 3*
  Dieses System ist eine an der ETH Zürich entwickelte Variante des ursprünglichen Oberon-Systems. Es bietet eine verbesserte grafische Benutzeroberfläche, unterstützt jedoch nicht die Programmiersprache Oberon-2 und daher auch keine typgebundenen Prozeduren. Die meisten Beispiele dieses Buches laufen daher unter diesem System nicht. Man kann dieses Oberon-System von *ftp.inf.ethz.ch/pub/software/Oberon/System3/* laden.

- *Weitere Oberon-Systeme und Compiler*
  Siehe *http://www.math.tau.ac.il/~guy/Oberon/*

Sonstige Oberon-Bücher

Neben dem vorliegenden Buch sind folgende Bücher zum Thema Oberon erschienen:

- *Martin Reiser, Niklaus Wirth: Programmieren in Oberon – Das neue Pascal. Addison-Wesley 1994.*
  Einführendes Programmierlehrbuch für die Sprache Oberon. Gut geeignet für Schulen und für das Selbststudium.

- *Martin Reiser: The Oberon System – User Guide and Programmer's Manual. Addison-Wesley 1991.*
  Beschreibt die Bedienung des Oberon-Systems sowie die wichtigsten Module der Oberon-Bibliothek.

- *Niklaus Wirth, Jürg Gutknecht: Project Oberon – The Design of an Operating System and Compiler. Addison-Wesley 1992.*
  Beschreibt die Implementierung des Oberon-Systems und des Oberon-Compilers.

- *Jörg Mühlbacher, Bernhard Leisch, Ulrich Kreuzeder: Programmieren mit Oberon-2 unter Windows. Hanser-Verlag 1995.*
  Ein einführendes Lehrbuch in die Programmierung mit Oberon-2 sowie eine Beschreibung der Programmierumgebung Pow! unter Windows.

- *E. W. Nikitin: Into the Realm of Oberon – An Introduction to Programming and the Oberon-2 Programming Language. Springer-Verlag 1998.*
  Ebenfalls ein einführendes Lehrbuch in die Programmierung mit Oberon-2.

Internet-Quellen

Am Internet findet man unter anderem folgende Informationen zum Thema Oberon.

- *news:prog.lang.oberon*
  News-Gruppe für Fragen und Diskussionen zum Thema Oberon.

- *http://www.oberon.ethz.ch/oberon/*
  Hauptseite der Oberon-Gruppe an der ETH Zürich.

- *http://www.ssw.uni-linz.ac.at/Oberon.html*
  Hauptseite der Oberon-Gruppe an der Universität Linz.

- *http://www.math.tau.ac.il/~laden/Oberon.html*
  Äußerst umfassende Informationen und Verweise auf Oberon-Systeme, Compiler, Werkzeuge, Bibliotheken sowie auf Literatur und Neuigkeiten rund um das Thema Oberon. Die Informationen wurden von *Guy Laden* an der Universität von Tel Aviv zusammengestellt.

- *http://www.oberon.ethz.ch/oberon/education.html*
  Kontaktadressen von Universitäten und Schulen, die Oberon in der Ausbildung verwenden.

Literatur

[Abb83] Abbott R.: Program Design by Informal English Descriptions. Communications of the ACM, vol. 26 (11), 1983

[BDMN79] Birtwistle G.M., Dahl O.-J., Myhrhaug B., Nygaard K.: Simula Begin, Studentlitteratur, Lund, Sweden, 1979

[BeC89] Beck K., Cunningham W.: A Laboratory for Teaching Object-Oriented Thinking. Proceedings OOPSLA'89. SIGPLAN Notices vol.24 (10), 1989

[BMRSS96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: A System of Patterns – Pattern-Oriented Software Architecture. Wiley, 1996

[Boo91] Booch G.: Object-Oriented Design with Applications. Benjamin Cummings, 1991

[Bud91] Budd T.: Object-Oriented Programming. Addison-Wesley, 1991

[Cha92] Chambers C.: The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. Ph.D. thesis, Stanford, 1992

[Deu89] Deutsch P.: Design Reuse and Frameworks in the Smalltalk-80 System. In Biggerstaff T.J., Perlis A.J. (ed.): Software Reusability, Volume 2, ACM Press, 1989

[DoD83] Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), United States Departement of Defense, Washington D.C., 1983

[Fow97] Fowler M.: UML Distilled, Applying the Standard Object Modeling Language, Addison-Wesley, 1997

[GHJV96] Gamma E., Helm R., Johnson R., Vlissides J.: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 1996.

[GWM88] Gamma E., Weinand A., Marty R.: ET++ – An Object-Oriented Application Framework in C++. Proceedings OOPSLA'88, SIGPLAN Notices vol.23 (11), 1988

[GoR83] Goldberg A., Robson D.: Smalltalk-80, The Language and its Implementation. Addison-Wesley, 1983

[Hof90] Hoffman D.: On Criteria for Module Interfaces. IEEE Trans. on Software Engineering, vol.16 (5), 1990

[JeW74] Jensen K., Wirth N.: Pascal User Manual and Report, Springer-Verlag, 1974

[JoF88] Johnson R.E., Foote B.: Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988, pp. 22-35

[KrP88] Krasner G., Pope S.: A Cookbook for Using the MVC User Interface Paradigm in Smalltalk. Journal of Object-Oriented Programming Aug./Sep. 1988

[Mey86] Meyer B.: Genericity versus Inheritance. Proceedings OOPSLA'86, SIGPLAN Notices, vol.21 (11), 1986

[Mey87] Meyer B.: Object-Oriented Software Construction. Prentice Hall, 1987

[MLK95] Mühlbacher J.R., Leisch B., Kreuzeder U.: Programmieren mit Oberon-2 unter Windows. Hanser-Verlag, 1995

[Nik98] Nikitin E. W.: Into the Realm of Oberon – An Introduction to Programming and the Oberon-2 Programming Language. Springer-Verlag, 1998

[Par72] Parnas D.L.: On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, 15, 12, December 1972, pp. 1053-1058

[Pre95] Pree W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995

[Pre96] Pree W.: Framework Patterns. SIGS Books & Multimedia, 1996.

[RBPEL91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design. Prenctice Hall, 1991

[Rei91] Reiser M.: The Oberon System; Users Guide and Programmers Manual. Addison-Wesley, 1991

[ReW92] Reiser M., Wirth N.: Programmieren in Oberon – Das neue Pascal. Addison-Wesley, 1994

[RGJ98] Rumbaugh J., Booch G., Jacobson I.: Unified Modeling Language Reference Manual. Addison-Wesley, 1998

[Sch86] Schmucker K.J.: Object-Oriented Programming for the Macintosh. Hayden, 1986

[Sed88] Sedgewick R.: Algorithms. Addison-Wesley, 1988

[Str86] Stroustrup B.: The C++ Programming Language, Addison-Wesley, 1986. Second Edition 1991

[Str89] Stroustrup B.: Multiple Inheritance for C++. Proceedings EUUG Spring Conference, Helsinki, May 1989

[Swe85] Sweet R.E.: The Mesa Programming Environment. SIGPLAN Notices vol.20 (7), 1985

[Szy92] Szyperski C.A.: Write–ing Applications. Proceedings of Tools Europe 92, Dortmund, 1992

[Web89] Webster B.F.: The NeXT Book. Addison-Wesley, 1989

[WiG92] Wirth N., Gutknecht J.: Project Oberon. The Design of an Operating System and Compiler. Addison-Wesley, 1992

[Wir71] Wirth N.: Program Development by Stepwise Refinement. Communications of the ACM, Vol.14, No. 4, 1971

[Wir82] Wirth N.: Programming in Modula-2. Springer-Verlag, 1982

[WiW89] Wirfs-Brock A., Wilkerson B.: Variables Limit Reusability. Journal of Object-Oriented Programming, May/June 1989, pp. 34-40

[WWW90] Wirfs-Brock R., Wilkerson R., Wiener L.: Designing Object-Oriented Software. Addison-Wesley, 1990

Index

A

Abbott, Methode von 151
ABS 278
abstract factory 112
abstrakte Datenstruktur 38
abstrakte Klasse 75, 91, 99, 102, 106, 155, 164, 168, 197
abstrakte Methode 76, 135
abstrakter Datentyp 7, 41, 89
Abstraktion 11, 29, 156, 251
Adapter 117, 249
ADR 285
ADS 38
ADT 41
aktueller Parameter 24, 270, 276
Algorithmenentwurf 146
Alias-Import 280
Änderbarkeit 4, 6, 100, 154
Anforderungsdefinition 148
Anpassen von Algorithmen 136
Anpassen von Klassen 103
Anpassen von Klassenschnittstellen 117
Anweisung 22, 269
Anweisungsfolge 271
Anwendungen von OOP 89, 108
application framework 180, 182
arithmetischer Ausdruck 21
arithmetischer Operator 267
Array 19, 262
Arrayindizierung 266
Arraykompatibilität 276, 283
AsciiTexts 200
ASH 278
ASSERT 278
Assertion 82
Attribut 45
aufgabenorientierter Entwurf 145
Ausdruck 21, 265
Ausdruckskompatibilität 21, 267, 283
austauschbares Verhalten 101, 129
automatische Speicherbereinigung 17, 20, 288

B

Backus-Naur-Form 257
Basisklasse 57
Basistyp 7, 263, 282
Basistyptabelle 289
befreundete Klassen 50
Benutzereingaben 197
Benutzt-Beziehung 67, 70, 161, 168
Beobachter 133, 174
Bezeichner 265
Beziehungen zw. Klassen 67
bindender Lader 32
Binärbaum, generischer 91
BIT 285
Block 24, 260
Blättern in Textrahmen 216
BOOLEAN 18
Boolescher Ausdruck 22
Boolescher Operator 267
Bottom-up-Entwurf 146
broadcast 129, 176, 190
Browser 26, 254, 289
BYTE 285

C

C 28, 97
C++ 20, 64, 65, 70, 94
CAP 278
CASE 22, 271
CC 285
CHR 278
class responsibility collaboration 152
composite 118
Controller 173
COPY 278
CRC-Karte 152

D

Datenabstraktion 7, 89, 130
Datenkapsel 4, 40
Datentyp 18, 261
davorgeschaltetes Verhalten 120
DEC 278
Definitionsmodul 26
Deklaration 259
Dekorator 119
derselbe Typ 282
Design 145
design pattern 109
designator 265
Dialogprogramm 180
DIV 267
Dokumentation von Klassen 253
Draw0 242
Durchlaufen einer Objektmenge 130
dynamische Bindung 8, 74, 78, 278
dynamische Erweiterung 143
dynamische Speicherallokation 20, 264
dynamische Typprüfung 127
dynamischer Typ 61, 112, 114, 137, 265
dynamischer Typ von Var-Parametern 64
dynamisches Array 20, 24, 53
dynamisches Laden 32, 100, 144, 180, 288

E

early binding 74
EBNF 257
Edit0 230
Effizienz 41, 254
Eiffel 94
Eigenschaften objektorientierter Sprachen 6
Ein/Ausgabe von Objekten 140
einfacher Datentyp 18
Einfachheit 154
Eingriffspunkt in Algorithmen 136
Element 179, 206, 230
Element-Metrik 208
Elementarität 154, 155
Elementtyp 262
Empfänger 46, 47, 51, 54, 163
Empfängerparameter 275, 276, 277
Endfabrikat 7
ENTIER 278
Entladen von Modulen 32, 287
Entwurf 145
Entwurf von Frameworks 168
Entwurfsfehler 159
Entwurfsmuster 109
Entwurfsüberlegungen 150
Ereignisdiagramm 134
ereignisgesteuertes Programm 181
Erweiterbarkeit 100, 119, 251
Erweiterbarkeit in mehrere Richtungen 121
Erweiterung 7
Erweiterung zur Laufzeit 100, 143, 180
Erweiterungsrichtungen, orthogonale 121
erzeugendes Muster 111
Erzeugung von Objekten 111, 112, 114, 138, 140
ET++ 182
EXCL 278
EXIT 273
Export 25, 280
Export von Methoden 278
Export von Recordfeldern 263
Exportmarke 260

F

Fabrik 112, 115
factory 112
Fallstudie 185
Fallunterscheidung 5, 98, 100
Familie 116, 117
Feld 19, 263, 266
Fenstersystem 186
Flexibilität 161, 254
flexible Objekterzeugung 112, 114
Focus-Rahmen 189, 197
font 206
FOR 22, 272
formaler Empfänger 47
formaler Parameter 24, 270, 275, 276
Fortran 28
Frame 187
Frame-Koordinaten 188
Framework 96, 167
Framework versus Prozedurbibliothek 169
Framework-Entwurf 168
Friend-Konstrukt in C++ 50
frozen spot 167
frühe Bindung 74
Funktionsprozedur 25, 274, 275, 276

G

ganzzahliger Typ 262
gap text 202
garbage collection 17, 20, 288
Geheimnisprinzip 6, 20, 39, 41, 49
generische Operation 132
Generizität 91
Gerüst 167
Geschichte 14
getrennte Übersetzung 28
gleiche Typen 282
Grafikeditor 232
Grafikelement 245
Grammatik von Oberon-2 284
GraphicElems0 246
GraphicFrames0 238
Gruppierung von Objekten 118
Gültigkeitsbereich 24

H

Halbfabrikat 7, 106, 167
Halde 35
HALT 278
Hat-Beziehung 68, 161
Hauptschleife 197
heap 35
heterogene Datenstruktur 95, 97
Hinzufügen v. Eigenschaften 119
Hollywood-Prinzip 170, 181
homogene Datenstruktur 95
hook 136
hot spot 167, 215
hybride objektorientierte Sprache 14

I

IF 22, 271
Import 25, 280
Import, zyklischer 29
In 32, 292
INC 278
INCL 278
Index in Arrays 19
information hiding 6, 20, 39
Inhaltsrahmen 187
Initialisierung von Objekten 111
Initialisierung von Modulen 29
INTEGER 18
Interaktionsdiagramm 134
interaktives Programm 173, 180
Internet-Adressen 316
Ist-Beziehung 59, 68, 161, 162
Iterator 130

J

Java 17, 64, 129
Jojo-Effekt 254

K

Klasse 9, 45
Klassenassertion 82
Klassenbibliothek 253

Klassendiagramm 45
Klassenentwurf 148
Klassenhierarchie 59, 116, 164
Klasseninvariante 83, 85
Klassenschnittstelle 50, 95, 153
Klassensuche 148
Klient 28
kombinierte Erweiterungen 121
Kommando 30, 144, 228, 280, 286
Kommandoparameter 31, 287
Kommentar 259
Kommunikation 11
Kompatibilität 8
Kompatibilität in Ausdrücken 21, 267, 283
Kompatibilität zwischen Klassen 59, 60, 117
Komplexität 35, 90, 158, 251
Kompositum 118, 197
konkrete Datenstruktur 35, 38
konkrete Klasse 164
konkreter Datentyp 7
konsistente Sichten 133, 173
Konsistenz 153
Konstantenausdruck 261
Konstantendeklaration 261
Konstruktor 111
Kontrakt 81
Kontravarianz 87
kooperatives Multitasking 198
Koordinatensystem in Grafikrahmen 238
Kopieren von Objekten 137
Korrektheit einer Klasse 83, 85
Kosten der Datenabstraktion 90
Kosten einer Klasse 153
Kosten von OOP 251
Kovarianz 88
Kurzschlußauswertung 22

L

late binding 74
Laufzeit-Typinformation 289
Laufzeiteffizienz 254
Laufzeiterweiterung 143
Laufzeittyp 138
Laufzeitumgebung 17
leere Anweisung 269
LEN 278
Lesbarkeit 4
logischer Operator 267
Lokalität 4, 100, 260, 275
Lokalität von Methoden 54
LONG 278
LONGINT 18
LONGREAL 18
LOOP 22, 273
LSH 285
Länge einer Zeichenkette 259
Länge eines Arrays 262
Länge eines offenen Arrays 276
Lückentext 202

M

MacApp 182
main event loop 197
Mausklick 197
Mausklick in Grafikrahmen 238
Mausklick in Textrahmen 216
Mauszeiger 197
MAX 278
mehrfache Vererbung 68, 123
Meldung 6, 9, 47
Meldungsinterpretation 9, 73, 78
Meldungsinterpreter 127, 177, 229
Meldungsobjekt 127, 132, 177, 190, 229
Mengenausdruck 22
Mengendiagramm 59
Mengenkonstruktor 268
Mengenoperator 268
Mengenzugehörigkeit 268
Menürahmen 187
Merkmale von OOP 15
Merkmale von OO Sprachen 6
Methode 9, 45, 277
Methodentabelle 129, 289
MIN 278
MOD 267
Model-View-Controller 173
Modul 25, 39, 280
Modul *Types* 138
Modula-2 17, 97
modulare Zerlegung 2
Moduldeskriptor 141

Module und Klassen 49
Modules 297
Modulinitialisierung 28
Modulkopplung 30
Modulrumpf 28, 280
Modulschnittstelle 289
Modus 181
Muster 109
MVC-Schema 173, 187, 199, 229
MVC-Schema in Oberon 175

N

Nachbedingung 81, 82, 85
Name 258
Namenskonflikt 69, 124
Namenskonventionen 154
Namespace in C++ 49
NEW 20, 264, 278
NextStep 182
NIL 19, 20, 264
Nonterminalsymbol 257
numerischer Typ 262
Nutzen von OOP 251

O

Oberklasse 57
Oberon-2 17, 257
Oberon-System 17, 286, 315
Oberon0 185, 198
Objekt 9
objektbasiert 2
objektorientierte Programmstruktur 4
objektorientierter Entwurf 146
observer 133, 174
ODD 278
offenes Array 24, 263, 264
Operator 259, 266
ORD 278
orthogonale Erweiterbarkeit 121
OS 299
Out 28, 295
overriding 58

P

Parameter 24, 276
Parameter überschriebener Methoden 87
Parameterliste 276
Parameterübergabe 276
parametrisierbare Operation 132
Parnas 7
Pascal 17, 35, 64
Persistenz 142
Pflichtenheft 148
Polymorphismus 2, 8, 47, 61
postcondition 81
precondition 81
Prioritätenschlange 35
priority queue 35
programming by difference 183
Programmrahmen 167
Projektion 63
Prototyp 114, 142, 242, 244
Prozedur 23
prozedurale Programmstruktur 3
Prozeduraufruf 270
Prozedurbibliothek 169
Prozedurdeklaration 275
Prozedurkopf 275
Prozedurrumpf 275
Prozedurtyp 20, 264
Prozedurvariable 9, 20, 48, 129, 132, 172
PTR 285
Pull-Modell 134
Push-Modell 134

Q

qualifizierter Name 28, 260

R

Rahmen 187
Rahmen-Metrik 219
Rautenstruktur 69
Read-Only-Variable 29, 39, 53, 260, 266
REAL 18
Record 19
Recordfeld 266
Recordtyp 263
Recordzuweisung 63, 270
Rectangles0 243

Redundanzfreiheit 154
reelle Zahl 258
reeller Typ 262
Referenzparameter 24
Rekursion 25, 275
REPEAT 22, 272
RETURN 25, 274, 275
Rollbalken 219
ROT 285

S

Sather 133
Schablonenmethode 77, 135, 170
Schleife 22
Schlüsselwörter 259
Schnittmenge 22
Schnittstelle 38
Schnittstelle überschriebener Methoden 87
Schnittstelle von Klassen 50, 77, 95
Schnittstelle von Modulen 25
Schnittstellenbrowser 26
Schnittstellenentwurf 153, 155
Schnittstellenprüfung 28
schreibgeschützte Variable 29, 39
Schreibschutz 260, 266
Schriftarten 206
schrittweise Verfeinerung 10, 146
scrolling 216
Selektion 229
semantische Lücke 11
SET 18
Set, Klasse 51
Shapes0 232
SHORT 278
short circuit evaluation 22
SHORTINT 18
Sicht 173
Sichtbarkeitsbereich 260
Simula 14
SIZE 278
Smalltalk 9, 14, 17, 64, 128, 133, 157
Speicherbedarf für Objekte 255
Speicherbereinigung 17, 20, 288
Spezialisierung 60
Sprachdefinition von Oberon-2 257
späte Bindung 74
Standardprozedur 22, 25
Standardtyp 262
statische Bindung 74
statischer Typ 265
Steckplatz 167, 215
Stilliste in Texten 207
Strichpunkt 271
String 259
Stringzuweisung 23, 270
strukturierter Typ 19, 261
Strukturierung von Programmen 29
Strukturmuster 116
Subkontrakt 83
Subsystem von Klassen 49
Symboldatei 28
Syntax von Oberon-2 284
SYSTEM 285
System.Free 287
systemnahe Programmierung 285

T

teilweise abstrakte Methoden 77
template method 135
Terminalsymbol 257
Terminologie 9
Textdarstellung 215
Texteditor 199
Textelement 179
TextFrames0 215
Texts0 206
tiling viewers 186
Toolbox 169
Top-down-Entwurf 146
Traversieren einer Objektmenge 130
triviale Klasse 160
Typdeklaration 261
Typdeskriptor 255, 289
type cast 285
type guard 65, 266
Typeinschluß 262, 282
Typenprüfung, strenge 18
Typerweiterung 55, 263, 264, 282
Types 138, 298

typgebundene Prozedur 46, 264, 276, 277
Typinformation 138, 289
Typname 138, 140
Typtest 64, 127, 268, 274, 290
Typumwandlung 65
Typzusicherung 65, 132, 266, 274

U

übereinstimmende Parameterlisten 264, 278, 284
Überschreiben von Methoden 58, 76, 278
Übersetzungseinheit 30
UML-Notation 45, 75
unabhängige Übersetzung 28
unbekannte Meldung 129
Unterklasse 57

V

VAL 285
Val-Parameter 24, 270, 276
Var-Parameter 24, 270, 276
Variablendeklaration 265
Varianten von Objekten 97
Varianten-Record 97
Verbund 19
Vereinigungsmenge 22
Vererbung 7, 55
Vergleichsausdruck 21
Vergleichsoperator 268
Verhalten, austauschbares 101, 129
Verhaltensmuster 126
Verschieben von Fenstern 195
Verzweigung 22
Viewers0 187
virtuelle Sprache 13
Vorausdeklaration einer Prozedur 276, 278
Vorausdeklaration eines Zeigerbasistyps 260
Vorbedingung 81, 82, 85
vordeklarierte Prozedur 264, 278
vordeklarierter Name 260
Vorrangregeln für Operatoren 266

W

Wert einer Klasse 153
Wertebereich von Standardtypen 18, 262
Werteparameter 24
WHILE 22, 272
Wiederverwendbarkeit 154, 252
Wiederverwendung 8, 10, 167
Wiederverwendung von Design 156
Wirth 17
WITH 66, 127, 139, 274
wrapper 118

X

Xerox 14

Z

Zahl 258
Zeichen-Deskriptor 228
Zeichen-Metrik 219, 228
Zeichenkette 259
Zeichenkettenvergleich 21
Zeichenkettenzuweisung 23, 270
Zeichenkonstante 259
Zeiger-Basistyp 264
Zeiger-Dereferenzierung 266
Zeigertyp 19, 264
Zeilen-Deskriptor 220, 228
Zeilen-Metrik 219, 228
Zugriffsprozedur 38
Zählschleife 272
zusammengesetztes Objekt 118
Zustand 11, 39, 181
Zustandsänderung 134
Zuweisung 269
Zuweisungskompatibilität 60, 269, 282
Zwilling 123
zyklischer Import 29, 280

Springer und Umwelt

Als internationaler wissenschaftlicher Verlag sind wir uns unserer besonderen Verpflichtung der Umwelt gegenüber bewußt und beziehen umweltorientierte Grundsätze in Unternehmensentscheidungen mit ein. Von unseren Geschäftspartnern (Druckereien, Papierfabriken, Verpackungsherstellern usw.) verlangen wir, daß sie sowohl beim Herstellungsprozess selbst als auch beim Einsatz der zur Verwendung kommenden Materialien ökologische Gesichtspunkte berücksichtigen.
Das für dieses Buch verwendete Papier ist aus chlorfrei bzw. chlorarm hergestelltem Zellstoff gefertigt und im pH-Wert neutral.

Springer

Hanspeter Mössenböck

Objektorientierte Programmierung in Oberon-2

3., vollständig überarbeitete und erweiterte Auflage

Der Autor führt den Leser von den Grundlagen objektorientierter Programmierung über Entwurfs- und Codierungstechniken hin zu einer realistischen Fallstudie in Form eines objektorientierten Fenstersystems mit Text- und Grafikeditor. In UML-Notation wird gezeigt, wofür sich objektorientierte Programmierung eignet und welche Probleme man mit ihr lösen kann. Als Programmiersprache wird Oberon-2 verwendet, ein moderner Nachfolger von Pascal. Die dritte Auflage enthält neue Kapitel über Entwurfsmuster und Kontrakte, ein erweitertes Kapitel über Frameworks sowie ein Glossar.

EXTRA MATERIALS extras.springer.com

Die beigefügte CD-ROM enthält Oberon-2-Compiler für Windows 95/NT, PowerMac und LINUX. Weitere Oberon-2-Compiler sind von der ETH Zürich erhältlich.

Hanspeter Mössenböck
ist Professor für Informatik an der Johannes Kepler Universität Linz. Seine Arbeitsgebiete sind objektorientierte Programmierung, Compilerbau und Werkzeuge der Softwaretechnik. Als ehemaliger Mitarbeiter von Professor Niklaus Wirth an der ETH Zürich war er an der Entwicklung der Sprache Oberon-2 beteiligt.

ISBN 978-3-540-64649-5

9 783540 646495

▶ **Software-Engineering**

▶ Programmierer/Studenten

http://www.springer.de